1-1-1975

An integrated ibm system/360 simulator and macro assembler for the cdc-6400.

Thomas Alan Salter

AN INTEGRATED IBM SYSTEM/360 SIMULATOR AND MACRO ASSEMBLER FOR THE CDC-6400

by

Thomas Alan Salter

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

1975

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

December 5, 1975
(date)

Professor in Charge

Chairman of Department

Table of Contents

Abstract

During the 1974-75 year three programs were written for use in EE 315, Principles of Computer Software. The three, an IBM 360 simulator, an IBM 360 assembler, and a general purpose macro processor, enabled students to obtain experience programming in IBM 360 assembly and machine languages. As the programs were used, a number of drawbacks were noticed and enhancements suggested.

This thesis describes the work that was done to the three to improve their operating performance and to enhance the power of the system. In particular, the user interface of the simulator was redesigned to make it simpler to use, and the macro processor was combined into the first pass of the assembler.

1.0 Introduction

This thesis describes the modifications and additions that were made during the fall of 1975 to the Lehigh University IBM 360 Simulator (LUIS) and the Lehigh University IBM 360 Assembler (LUIAS).

These two programs and a third, a macro processor, were used extensively for the first time by the Spring 1975 EE 315 class. Their use in this class brought about a number of suggestions for improvements to the programs. In particular, the complaints were that LUIS required too many type-ins by the user and that the free-standing macro processor differed too much from the IBM macro assembler. In addition, a number of additional pseudo-operations were suggested for LUIAS.

Part two of this thesis describes the work done on LUIS. A new user's manual is included, along with flowcharts and descriptions of the new routines.

Part three describes the combination of the macro processor into the first pass of LUIAS. It also describes the addition of CSECT, DSECT and CNOP pseudo-instructions. A user's manual provides a complete description of how to use LUIAS. Also included are a comparison of LUIAS with the IBM assembler and detailed descriptions and flowcharts of the LUIAS routines.

The four appendices describe the character set used by both LUIS and LUIAS, the operations assembled by LUIAS,

the load and compilation sequence for LUIAS, and sample programs.

This thesis does not go into detailed descriptions of code and routines which were not modified by the author. Complete descriptions of the work done previously are available in the master's theses of Horey (2), Maroudas (4), and Seager (5). These three describe LUIS, the macro processor, and LUIAS, respectively.

The source code for both LUIS and LUIAS is available from the Lehigh University Computing Center.

2.0 The Lehigh University IBM 360 Simulator

This section describes the modifications made to the Lehigh University IBM 360 Simulator (hereafter referred to as LUIS or the simulator).

Section 2.1 is the user's manual for the simulator. It supplies the needed information for a user of the simulator. It describes the commands and the differences between the simulator and a real IBM 360.

Section 2.2 describes the motivation behind the modifications to the simulator. It also explains why certain strategies were employed in the implementaion of various functions.

Section 2.3 provides a detailed description of the changes made to the simulator. Included are detailed flowcharts showing the logic flow within the added subroutines.

2.1 User's Guide to the IBM 360 Simulator

This section is a reproduction of the user's manual which will be made available to the users of this simulator. Because it is intended for the user, it contains no references to the code or the internal routines of the simulator. Instead, it contains all the information that a user of the simulator needs to know in order to make efficient use of the simulator, providing of course, that the user is familiar with IBM 360 machine language. In particular, this section describes the command formats and the functions of these commands.

It should be noted that this manual is a revision of a similar manual prepared by Horey (2) for an earlier version of the simulator.

Lehigh University IBM Simulator

User's Guide

Introduction

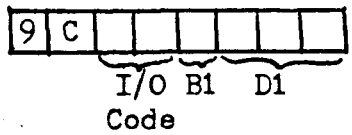
The Lehigh University IBM Simulator (LUIS) executes a program written in IBM System/360 machine language on a Control Data Corporation 6400 Computer System. LUIS simulates all but seven of the instructions in the IBM System/360 standard instruction set. (The instructions Diagnose, Set System Mask, Load PSW, Halt I/O, Supervisor Call, Test Channel, and Test I/O are omitted.)

One important difference between the simulator and the IBM 360 is that the maximum address is specified in 17 bits rather than 24 bits. If the eighteenth bit is a one, the address will be interpreted as a negative number and LUIS will terminate the program if and when that address is actually used for addressing. If more than 18 significant bits are used, the address will be truncated when it is used for addressing. If the truncated address is within the memory area of the program, the program will execute with that address. However, if the resulting address is outside the memory area of the program, the simulator will issue an error message and terminate the program. This restriction on the length of addresses is a result of using the 18-bit A and B registers in the CDC 6400 for computing addresses. (One should note that if an address field is not actually used for addressing storage, as in Load Address and shifting instructions, this limitation does not apply.)

The Instruction Set

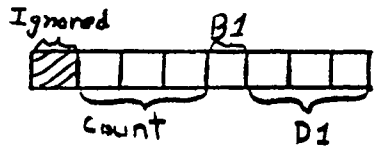
All of the simulated instructions (except Start I/O) function as described in the manual, IBM System/360 Principles of Operation. The floating-point feature, decimal feature, protection feature and direct control feature instructions are not available. Conditions which would normally produce an interrupt in an IBM 360 cause the simulator to terminate execution of the program and to print an error message indicating the cause of the interrupt.

A modified version of the Start I/O (SIO) instruction allows the user's program to perform I/O. The internal format of this instruction is as follows:



The operation code is hexadecimal '9C'. The I/O code in the instruction indicates the type of I/O operation that is to be performed by the program. The code is '00' for reading and '01' for writing in hexadecimal dump format, and it is '02' for reading and '03' for writing in EBCDIC character format. All other codes are invalid. The address specified by B1 and D1 specifies an I/O control word located somewhere in the user's (memory) storage area.

The I/O control word is a fullword which contains the count of the number of words to transfer and the starting address of the first word. Its format follows:



The count is the number of words which will be transfered. If the count is zero, no I/O operation occurs. The address specified by B1 and D1 specifies the first fullword to be transfered.

Hexadecimal I/O operations are best used for debugging. On output, the address and its contents will be printed for each word transferred. On input, the simulator will print a message requesting the user to enter the required words for each address as printed.

Character I/O is most like I/O done by an IBM 360. On output, only the words specified are printed in character format; no messages are printed by the simulator. Characters are displayed 80 to a line, with each SIO command starting a new line. On input, the characters are accepted as typed; no preliminary message is printed. It is the responsibility of the 360 programmer to include a message informing the user that input is required. If no input is entered the simulator will simply wait, giving no indication that it is running. (This condition may be detected by using the INTERCOM type-in of '%C' to get the status of the program. A response of "WAITING FOR INPUT" indicates that the program is waiting for the user to type something.)

Any of the 64 valid characters in the SCOPE character set may be entered in a character string. They will be translated to the appropriate EBCDIC code and stored in hexadecimal. (See

Appendix A for the translation table.) When character strings are printed, the most significant two bits are ignored. This procedure reduces the 256 possible bit configurations of a byte to 64 which can be translated to SCOPE display code. Thus, a data error cannot occur during character I/O.

Use of the Simulator

LUIS is an interactive program which simulates the operation of an IBM 360 computer. To use the simulator, the user must attach it while in INTERCOM command mode:

ATTACH(LUIS,ID=xxx)

If the user has his 360 object program on a permanent file, it must be attached with a local name of PGM,

ATTACH(PGM,USERPROG,ID=xxx)

The simulator may then be executed with the following command:

LUIS.

When the simulator is loaded, it will type out the following:

LEHIGH UNIVERSITY IBM 360 SIMULATOR

REQUEST=

The user may enter any simulator command. The following list gives a brief description of each command; a more thorough description follows.

BYE - return to INTERCOM command mode.

DUMP - display 360 memory region.

END - terminate current 360 program.

EXECUTE - begin execution of 360 program.

HELP - display list of commands.

INSERT - load bytes into 360 memory.

MEMSIZE - specify size of 360 memory region.

PAUSE - specify address to pause 360 program.

PSW - specify 360 PSW.

REMOVE - remove effects of pause.

RESUME - continue execution after a pause.

REWIND - rewind PGM or STORE file.

STEP - execute a specified number of 360 instructions.

STATUS - display 360 PSW and general purpose registers.

STORE - store contents of 360 memory region on file.

The simulator operates in two modes. In REQUEST mode, the user may enter any of 15 commands which allow the user to control the operation of the simulated program. In the second mode, the simulator is under control of the simulated 360 program.

In REQUEST mode the user may enter commands to define the memory size of the 360 program, to enter and display bytes in the 360 memory region, and to control the execution of the program.

Once the simulated program has begun execution, the user gets control only if the program terminates or if the program does I/O. The program terminates if the specified number of instructions has been executed, if a user defined pause is reached, or if an error condition causes an interrupt. When the program does I/O, it is the user's responsibility to enter the required data. as needed.

Command Descriptions

All commands are entered in LUIS REQUEST mode; that is, after the simulator has typed:

REQUEST =

Commands are of variable length, not to exceed 39 characters. Any leading blanks are ignored. Commas and blanks are interchangeable as delimiters, except in specifying omitted parameters. Omitted parameters must appear as two consecutive commas separated only by spaces. Thus,

```
INSERT PGM,,200
```

specifies that 200 bytes are to be loaded at the default location generated by the assembler, but

```
INSERT PGM  200
```

specifies that as many bytes as the assembler generated are to be loaded beginning at the hexadecimal location 200.

Numeric values may be specified to the simulator with less than the maximum number of digits. The simulator assumes leading zeros to obtain the required number of digits. Thus, a length of 100 required as a four digit decimal number may be entered as either 100 or 0100.

If any parameter is entered incorrectly or is omitted, the simulator requests the user to enter it. At this time, the user may also, at his option, reenter any subsequent parameters. For example, if the user types:

```
MEMSIZE 30000,200
```

the simulator would request the user to type a valid starting address. He could type:

```
3000
```

and let the length 2000 remain as entered, or he could change

both by typing:

3000,500

Whenever the simulator requests the user to enter a parameter to clarify an incompletely or incorrectly specified command, the user may cancel the command by typing $A. This abort type-in ($A) is analogous to the INTERCOM abort type-in (%A) which returns the user to COMMAND mode. The abort type-in must be entered exactly as $A. It must be entered in the first two positions of a line; no leading spaces are permitted. The abort type-in is illegal if the simulator is already in REQUEST mode; it is also not recognized if the simulated program is requesting input.

The remainder of this section describes in detail each of the simulator REQUEST mode commands.

BYE

This command terminates the simulator and returns the user to INTERCOM COMMAND mode.

DUMP ALL/PARTIAL,start,length

This command prints memory dump of a selected portion of the user's 360 program.

Positional Parameter 1:

ALL - causes a dump of the entire memory region (as specified by MEMSIZE)

PARTIAL - causes a dump of the portion of the memory region specified by parameters two and three.

Positional Parameter 2:

A hexadecimal number indicating the starting address of the area to be dumped. The address must specify a full word boundary.

Positional Parameter 3:

The length, in bytes, of the area to be dumped. It must be specified as a decimal integer.

If any of the parameters are omitted, the user will be requested to enter them as the simulator requires them.

END

This command clears the memory region for the 360 program and restores the default values for MEMSIZE and PSW.

EXECUTE

This command causes the 360 program to begin execution. If no error occurs, the program will cycle through 9,999 instructions and then stop, returning the user to REQUEST mode. If an interrupt occurs, the program's execution will be halted, an error message displayed, and the user returned to REQUEST mode.

HELP

This command displays a list of all the commands available in LUIS REQUEST mode.

INSERT INPUT/PGM/PGMSTORE,start,length

This command allows the user to enter all or part of his 360 program into the simulator's memory area.

Positional Parameter 1:

INPUT - indicates that the user is going to type the bytes as they are requested by the simulator.

PGM - indicates that the user's program resides on the file PGM as produced by the assembler (LUIAS). The starting address and length will be read from the file unless otherwise specified.

PGMSTORE - indicates that the user's program resides on the file PGM as produced by the simulator STORE command during a previous session. This option must also be used if the entire file produced by the assembler was not read originally.

Positional Parameter 2:

A hexadecimal number specifying the starting address at which the bytes will be loaded. The address must specify a fullword boundary.

Positional Parameter 3:

The number of bytes to be loaded, specified as a decimal integer.

Note:

Positional parameters two and three are optional if PGM is specified as parameter one. If specified, they will over-

ride the assembler produced starting address and length.

When reading file PGM, the entire file need not be read at once. However, to read only a portion of the file the length must be included in the INSERT command. Subsequent insertions should specify PGMSTORE instead of PGM and thus, must include the starting address and length of the bytes to be read.

MEMSIZE start,length

This command allows the user to define program limits defining the memory size of the IBM 360 being simulated. Any attempt by a user program to reference outside these limits causes an error and interrupt. If not specified, the default values are a starting address of zero and a length of 10,000.

Positional Parameter 1:

The starting address of the memory region. The value must specify a fullword address between zero and hexadecimal 1D8A8.

Positional Parameter 2:

The length of the memory region specified as a decimal integer. The length may not exceed 10,000 bytes.

PAUSE location

This command allows the user to specify an instruction address at which his 360 program is to pause. When the program

being executed reaches the instruction at the sepcified location, the simulator stops execution and types:

PAUSING...

REQUEST =

The user is returned to REQUEST mode and may enter any REQUEST mode command. In particular, he may print portions of his program, display the PSW and program registers, alter memory locations or change the PSW. To continue execution, the RESUME command should be used.

Positional Parameter 1:

The instruction address at which the program is to be paused. The specified location must be on a halfword boundary.

PSW word1,word2

This command allows the user to change the value of the PSW. The default value assumed by the simulator for the PSW is zero.

Positional Parameter 1:

The first word of the PSW as a hexadecimal number.

Positional Parameter 2:

The second word of the PSW as a hexadecimal number. The low order 24 bits of this word specify the address of the next instruction to be executed.

If either word of the PSW is omitted or entered incorrectly, both must be entered at the request of the simulator.

REMOVE location

This command removes the effects of a PAUSE command entered previously for the specified location. If there was no PAUSE at that location, an error message is printed.

Positional Parameter 1:

The location of the PAUSE to be removed.

RESUME

This command instructs the simulator to continue executing the user's program after a pause occurred. This command does not reset the instruction count as originally defined by STEP or EXECUTE. If STEP or EXECUTE are issued after a pause, the instruction count will be reset. The simulator only accepts this command when the user's program is pausing.

REWIND PGM/STORE

This command allows the user to rewind either file PGM or STORE without leaving the simulator. The user must rewind these files if he wishes to reload his program, or if he wishes to ensure that his saved program starts at the beginning of file STORE.

STEP number

This command allows the user to execute a specific

number of instructions. After execution of this number of instructions the user is returned to REQUEST mode.

Positional Parameter 1:

This number is a decimal number specifying the number of instructions to be executed. The maximum value permitted is 9,999.

STATUS

This command causes the simulator to print the PSW, the instruction address, the ILC, the condition code, and the contents of the 16 general purpose registers.

STORE ALL/PARTIAL,start,length

This command enables the user to dump a selected portion of the 360 program's memory region to a file named STORE. Files dumped in this manner may be read during a subsequent simulator run by using the INSERT PGMSTORE command, if the file has been copied from STORE to PGM between simulator runs. It is necessary to copy the file, because the simulator will only read from PGM and only write to STORE.

Positional Parameter 1:

ALL - dumps the entire memory region, as defined by MEMSIZE, to the file STORE.

PARTIAL - dumps only the portion of the memory region specified by positional parameters two and three.

Positional Parameter 3:

The address of the first fullword to be dumped.

Positional Parameter 3:

The number of bytes to be dumped specified as a decimal integer.

2.2 Technical Notes on Modifications to the Simulator

The Lehigh University IBM 360 Simulator was written during 1974-1975 by Leonard I. Horey. (2) Its use in the spring of 1975 by the EE 315 class uncovered several drawbacks to the simulator.

1) Long messages, up to 2 lines in length, took too much of the user's time when typed at a teletype. These messages were useful for the beginning user but were a nuisance for an experienced user of the simulator.

2) The only way to stop a simulated program once it started was to wait for an error to occur or until a preset number of instructions had been executed.

3) The only I/O available was in the form of hexadecimal dumps and hexadecimal type-ins. No character I/O was available. In addition to the program's I/O, the simulator also printed several messages telling the user that his program was doing I/O.

The original approach taken with regard to user-defined variables necessitated the use of many messages. For example, whenever a variable was needed by the simulator, the user was instructed to enter that number. Long messages were required to explain to a new user what values to enter.

The method suggested by Horey (2) for alleviating this inconvenience was to allow two sets of messages; full

messages for the new user; abbreviated ones for the experienced user. However, examination of the existing code revealed that this approach would be awkward to implement and would result in a larger and slower program.

Most of the messages and all of the user type-ins were handled by the main Fortran program, LUIS. This segment was short (less than 300 source code lines) and required major modifications, so it was decided to rewrite it rather than to attempt to patch it up. By rewriting, it was possible to do away with almost all of the long messages. Instead of forcing the user to define many parameters, they are given default values, and the user is provided commands to change these values whenever he wants. In addition, command formats were changed to allow parameters to be specified with the commands, rather than with a number of separate entries. A side effect of these changes was to make all commands free format.

For example, with the old simulator, dumping a part of memory required the following sequence. (Note: underlined portions are typed by the user; the remainder by the simulator.)

REQUEST =DUMP

TYPE ALL OR PARTIAL

PARTIAL

ENTER THE STARTING ADDRESS AS AN EIGHT DIGIT HEXADECIMAL NUMBER.

000014AC

ENTER THE NUMBER OF BYTES TO BE DUMPED
AS A FOUR DIGIT DECIMAL INTEGER.

0112

After that, the dump would be printed by the simulator. With the new simulator, all of this may be entered with the sigle type-in:

REQUEST =DUMP PARTIAL,14AC,112

However, if the user omits any of the parameters, the simulator will request them to be entered in much the same manner as the old one did. In fact, the old sequence listed above is still an acceptable one. One significant difference is that the leading zeros required by the old simulator are no longer needed. The user is also able to specify more parameters than the one he is requested to enter.

REQUEST =DUMP,PARTIAL

ENTER THE STARTING ADDRESS AS AN EIGHT DIGIT HEXADECIMAL NUMBER.

14AC,112

In the above sequence the first entry requests a partial dump; the second defines the starting address and the length in a single line.

Two subroutines, PARSER and PARSPRM, were written to handle input. PARSER searches a 39 character input string and splits off the command word and up to three parameters. Parameters are delimited by commas or spaces. All spaces and binary zeros preceeding a parameter are ignored. All parameters stored by PARSER are right-justified in a word with zeros filled to the left of

the parameter. (Compatible with Fortran R format) The subroutine PARSPRM is used when a parameter was omitted cr incorrectly specified. This subroutine uses PARSER to read the needed parameter. The command and the parameters preceeding the requested one remain unchanged. Parameters following the requested one are changed only if the user enters them along with the requested parameter.

These two subroutines provide the flexibility of input not found in the old simulator. Numbers need not be padded to a specific length, because spaces are treated as delimiters by PARSER and not as zeros as by Fortran FORMAT statements.

The old simulator would begin execution of a 360 program and not stop until an error occurred or until a specified number of instructions were executed. For debugging purposes, however, it is often convenient to pause a program before a specific instruction is executed. To this end, three instructions were added to the simulator: PAUSE, REMOVE, and RESUME.

PAUSE specifies an address at which the program is to pause. When this address is reached, the user is given control in REQUEST mode before the instruction is executed. While in REQUEST mode the user may use any simulator command. In particular, he may take dumps, examine the contents of registers, and change bytes in storage. To continue execution he enters the RESUME command. The REMOVE command is used to terminate

the effects of a pause, so that the program will no longer pause when the instruction at that address is executed.

The old simulator had no I/O options. The routines for reading and writing under program control were nearly identical to the INSERT and DUMP requests available to the user in REQUEST mode. Thus, there was no facility for testing the I/O formats of 360 programs. The new simulator contains, in addition to the old formats, the ability to read and write EBCDIC characters. In character mode, nothing is printed by the simulator except as requested by the user's program. Consequently, it is the 360 programmer's responsibility to inform the user that input is expected, just as it would be on a real IBM 360.

Input characters are translated from CDC-6400 display code to eight-bit EBCDIC. There existing no one-to-one correspondence of printable characters, the translation was made so that the Hollerith keypunch codes were kept constant. Inasmuch as there are 256 EBCDIC characters but only 64 CDC display codes, a decision had to be made with regard to the extra characters. Two possible solutions were to generate an error whenever one of the extra characters was encountered, or to translate all of the extra characters to spaces. The solution chosen was to ignore two of the eight bits in the EBCDIC code, leaving only six bits for 64 possible codes. This option alleviates the need for checking characters and causing a data

error when invalid characters are encountered.

Three subroutines were added to provide character I/O. Two Fortran subroutines, CHRITE and CHREED, do the actual I/O. These are similar to the hexadecimal routines, RITE and REED, found in the old simulator. The third, EBCDIC, a COMPASS subroutine, has two entry points for converting hexadecimal to display and display to hexadecimal. EBCDIC is called by CHRITE and CHREED..

Several other changes were made to make the simulator more convenient to use. Two commands were added, HELP and MEMSIZE; several were modified, END, INSERT, PSW, REWIND, and STEP.

The HELP command prints a list of the 15 command words available in REQUEST mode. Knowing only these keywords, the user can use all of the commands of the simulator. He need not know all of the parameter formats because the simulator will print prompting messages for each parameter.

The MEMSIZE command defines the starting address and length of the memory region. This command was not previously needed because the simulator forced the user to enter these values before going into REQUEST mode. The removal of the initialization messages necessitated the addition of a command to change the default values assumed by the simulator.

The END command formerly allowed the user to redefine his memory size by restarting the simualtor. Now it clears memory and restores the default values to MEMSIZE and PSW. It produces the same results as returning to SCOPE and recalling the simulator.

A change to the INSERT command was necessitated by changes to the assembler (LUIAS). The assembler now writes out the starting address and length of the assembled program, so that the user need not remember them. The PGM option of the INSERT command was modified to handle this case, but the user is still able to override the assembler produced values. A new INSERT option, PGMSTORE, was added to allow the user to read STORE files created by a previous simulator run. This ability, previously included in the PGM option, was lost when the format of the PGM file was changed. (See page 26 for simulator file formats.)

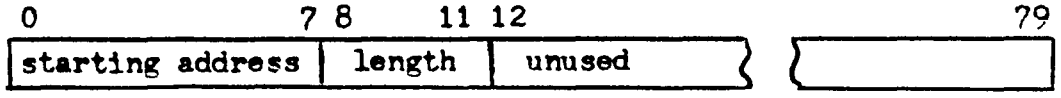
An additional change to the INSERT command affects the INPUT option. Words are checked for validity as they are typed, rather than after all have been typed. If an error is detected, the user is informed immediately and requested to enter a valid number. With the old simulator, the words were not checked until all had been read, and then, if an error was found, the entire command was aborted.

The old command of NEWPSW was replaced with PSW. When the old simulator was loaded, it asked the user to enter the PSW. NEWPSW was only used for changing the value of the PSW. NEWPSW required the entry of a 16-digit hexadecimal number. The PSW command in the new simulator must be used to change the PSW from its default value of zero. It requires two eight-digit hexadecimal numbers whose leading zeros may be omitted.

The old simulator had two commands, REWINDP and REWINDS, for rewinding the files PGM and STORE, respectively. These

PGM File

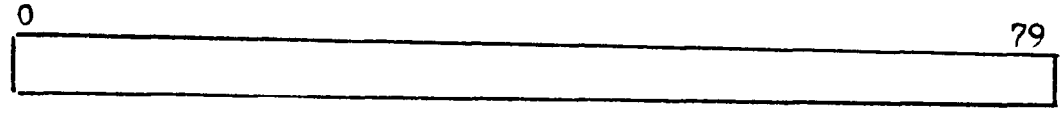
Record 1:



0-7 starting address - Eight display code characters specifying hexadecimal starting address of the program on this file.

8-11 length - Four display code characters specifying the number of bytes on the file as a decimal integer.

Records 2-N:



0-79 data - Eighty display code hexadecimal (0-9,A-F) characters corresponding to the contents of contiguous locations starting at the address specified in record 1.

STORE File

All records the same as records 2-N of file PGM.

Note: Whenever PGM file is positioned at a location other than record 1, it will appear to be identical to a STORE file.

commands were replaced with REWIND PGM and REWIND STORE, which are more resemblant of the SCOPE rewind commands.

The STEP command for executing a specific number of instructions replaces the old S command. The functions are identical, however STEP is more decriptive than S.

The new simulator reads all commands through the soubroutine PARSER, and all corrected input through the subroutine PARSPRM. This feature allcws the addition of an abort type-in which always returns the user to REQUEST mode. This type-in ($A) allows the user to stop a command after he has entered the command word, if he realizes that he has made a mistake. However, the abort type-in can only be entered when the simulator is expecting a parameter type-in. When PARSPRM detects the abort type-in, it calls the suboroutine ABORT which returns control to the main Fortran program at the REQUEST mode entry point. ABORT is a COMPASS subroutine which can execute the branch which the Fortran subroutine PARSPRM cannot.

The symbol $A was chosen for the abort type-in because of its similarity to the INTERCOM abort type-in %A. In INTERCOM, %A terminates the current command and returns the user to command mode; in the simulator, $A terminates the current command and returns the user to REQUEST mode. The primary difference is that INTERCOM can accept its abort type-in at any time, but the simulator can only accept its abort type-in when it is expecting input.

To reduce the load size if the simulator, the largest

common block (over 2500 words) was changed from labeled common to blank common. The SCOPE loader requires about 10k; however, blank common may overlay this region. Thus, the load size of the simulator was reduced to below 40k. This allows the simulator to be run on the Lehigh University Computing Center general purpose password, alleviating the need for a special account for users of the simulator.

2.3 Descriptions and Flowcharts of Modifications to the Simulator

This section provides a description of the workings of the various changes to the simulator. For the closed subroutines flowcharts are also included. The changes to FIEM involved only a few lines of code, so only a description of those changes is included. This section is intended to provide sufficient information for a person to make changes to the described code at some future date.

2.3.1 Changes to the Subroutine FIBM

The subroutine FIBM does the actual simulation of an IBM 360. Due to the size and complexity of this routine, changes to it were kept to a minimum.

A minor change was made to allow character I/O. When the I/O codes of two and three are detected in an SIO command, calls are made to the external subroutines CHREED and CHRITE.

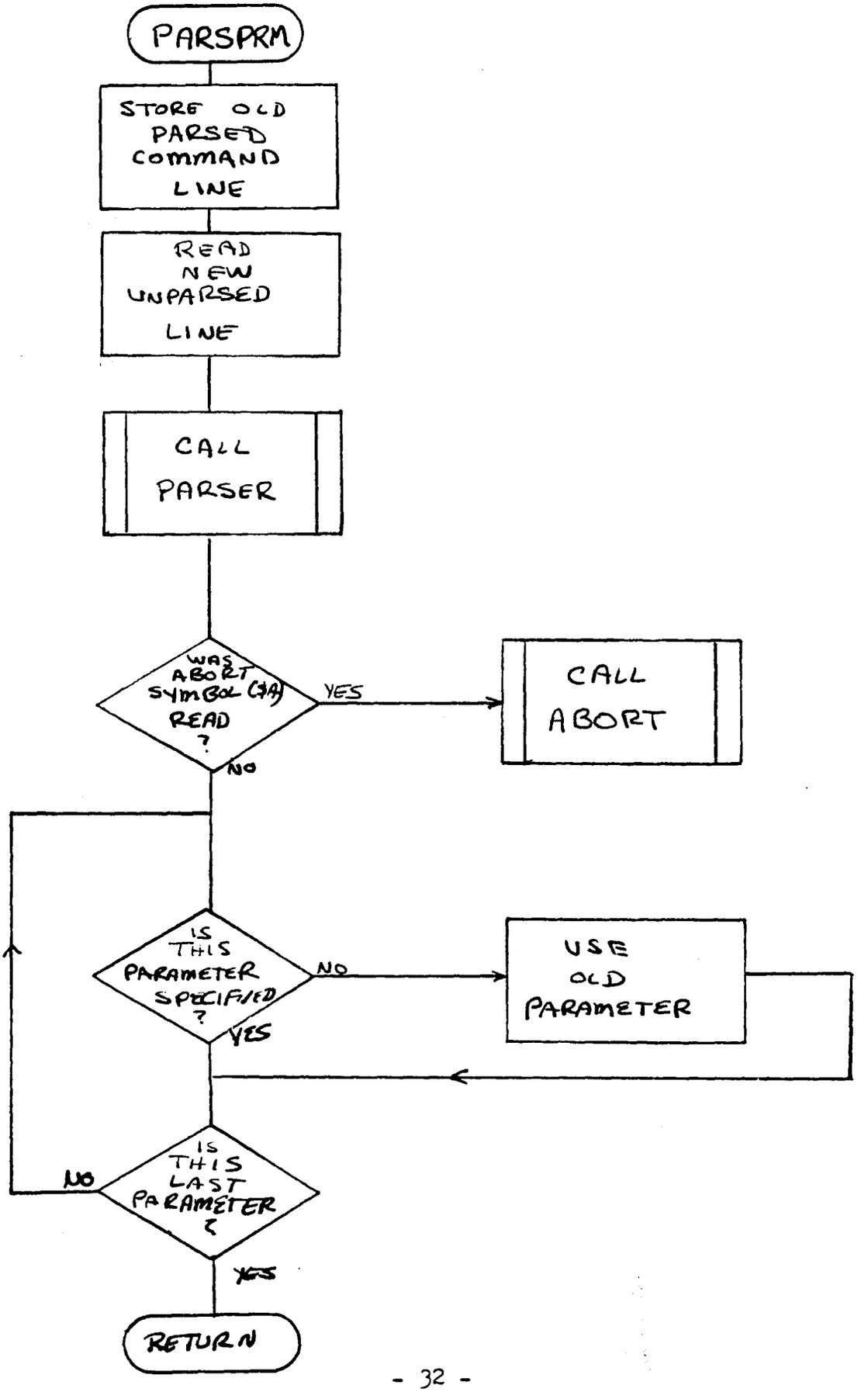
To allow program pauses, a change was made to the instruction fetch routine. Before an instruction is retrieved, its address is checked against those in the pause array. If a match is found, a flag is set and control is returned to the calling routine. To increase execution time only the used entries of the pause array are searched. Thus, only as many words as necessary are checked, even though the pause array is a constant 100 words long.

When FIBM is entered it checks to see if it is returning from a pause. If so, it skips the pause checking code. This test makes it possible to get past a pause.

2.3.2 Subroutine PARSPRM

The Fortran subroutine PARSPRM reads parameters as entered by the user. It first stores the parameters read by the original command. Then it creates dummy parameters to make the new entry appear to be the positional parameter requested of PARSPRM in its call. PARSPRM calls PARSER to split up the parameters in this line. The requested parameter always replaces the corresponding parameter of the orignal command. In addition, if any parameter is specified following the requested one, it also replaces the one originally entered.

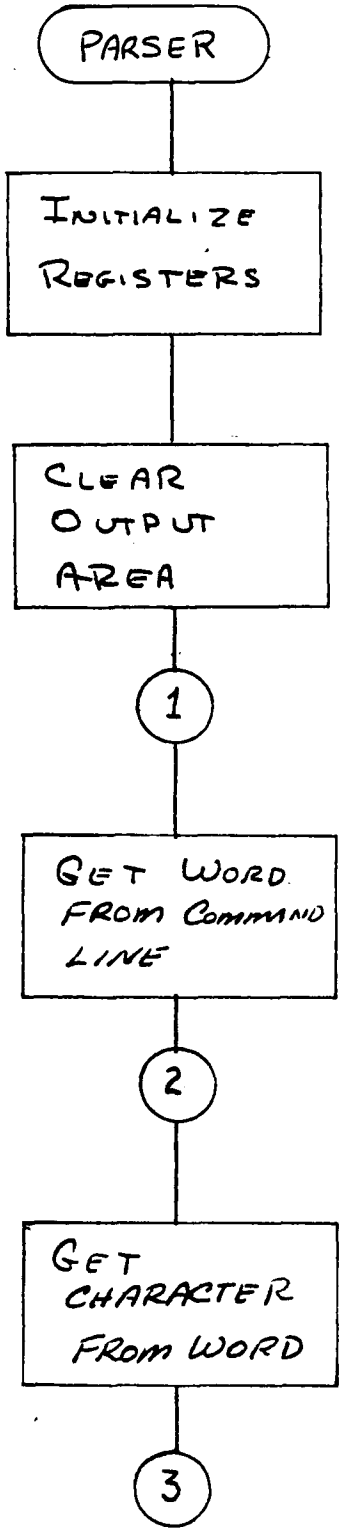
If the input entered by the user consists of the two characters "$A", PARSPRM calls subroutine ABORT to return the user to REQUEST mode immediately.

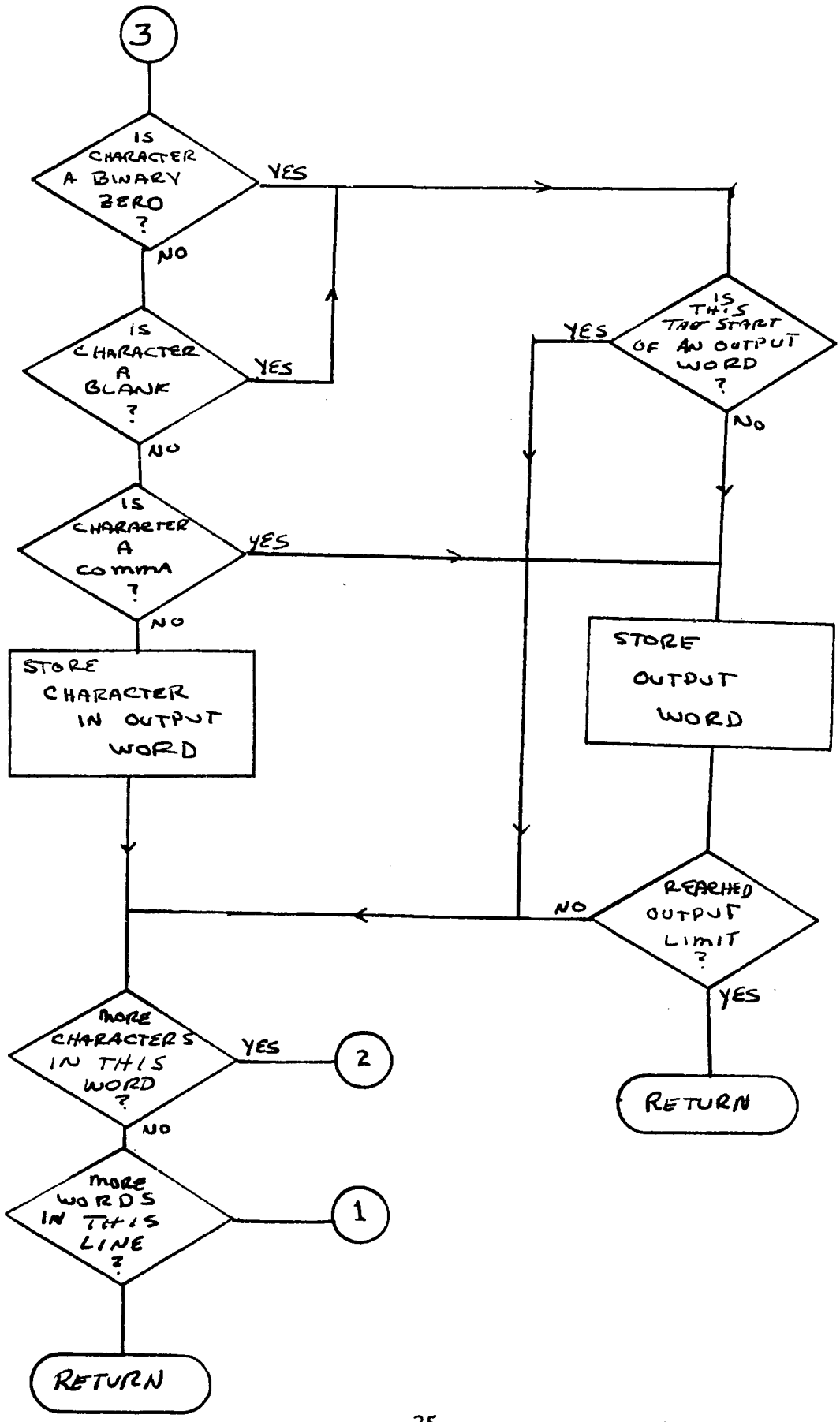

PARSPRM
STORE OLD PARSED COMMAND LINE
READ NEW UNPARSED LINE
CALL PARSER
WAS ABORT SYMBOL (^A) READ ?
YES
CALL ABORT
NO
IS THIS PARAMETER SPECIFIED ?
NO
USE OLD PARAMETER
YES
IS THIS LAST PARAMETER ?
NO
YES
RETURN

2.3.3 Subroutine PARSER

The COMPASS subroutine PARSER accepts the user's command as typed and separates the command and up to three parameters.

Binary zeros and blanks are ignored whenever found at the start of a parameter. All other characters are treated as part of a parameter. A parameter is terminated when the first binary zero, space, or comma is detected. Parameters may be up to ten characters long; if any are longer, the returned value will be unpredictable. The subroutine returns after the command word and three parameters have been found or when it reaches the fortieth input character. Input lines are limited to 39 characters because the fortieth character must always be blank. If the end of input is reached before three parameters are found, the remaining parameters are considered to be blank. A blank parameter may be indicated by two consecutive commas in the input line.


PARSER
INITIALIZE REGISTERS
CLEAR OUTPUT AREA
1
GET WORD FROM COMMAND LINE
2
GET CHARACTER FROM WORD
3

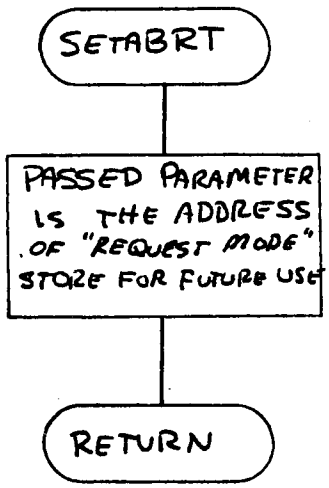

3
IS CHARACTER A BINARY ZERO ?
YES
NO
IS CHARACTER A BLANK ?
YES
NO
IS CHARARTER A COMMA ?
YES
NO
STORE CHARACTER IN OUTPUT WORD
IS THIS THE START OF AN OUTPUT WORD ?
YES
NO
STORE OUTPUT WORD
REACHED OUTPUT LIMIT ?
NO
YES
RETURN
MORE CHARACTERS IN THIS WORD ?
YES
2
NO
MORE WORDS IN THIS LINE ?
1
RETURN

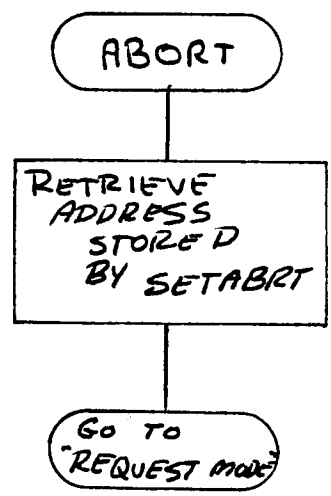
2.3.4 Subroutine ABORT

The COMPASS subroutine ABORT allows the Fortran subroutine PARSPRM to effect a non-standard return to the main Fortran routine.

The routine ABORT has two entry points. The first, SETABRT, is called only once during the execution of the simulator. With this call, it is passed the address of the REQUEST mode code in the main program. This value is saved for future use. The second entry point, ABORT, is called by PARSPRM when it reads the abort type-in ($A). ABORT retrieves the previously stored address and branches to the REQUEST mode code in the main program.

The use of this subroutine was the only reasonable way to allow any command to be terminated once the simulator had begun to process that command. To have used standard Fortran procedures would have required testing a flag in the main program after every call to PARSPRM. Inasmuch as there are at least a dozen calls to PARSPRM, using individual tests would have resulted in longer and more confusing code.


SETABRT
PASSED PARAMETER IS THE ADDRESS OF "REQUEST MODE" STORE FOR FUTURE USE
RETURN


ABORT
RETRIEVE ADDRESS STORED BY SETABRT
GO TO "REQUEST mode"

2.3.5 Subroutine CHRITE

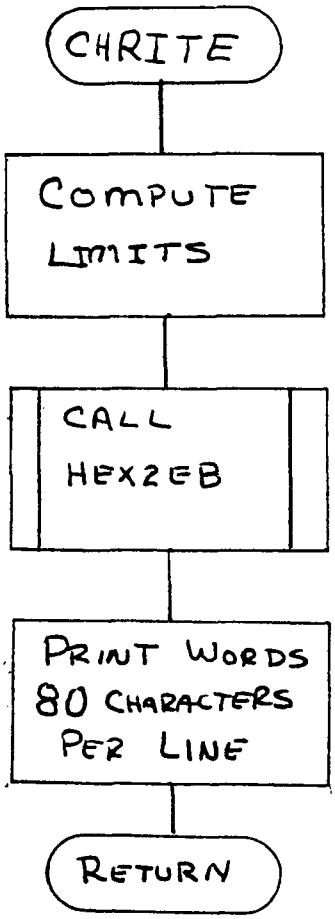
The Fortran subroutine CHRITE handles a request by the user's program to write in character format. Subroutine FIBM passes it three parameters: the first byte address, the corresponding index in the NW array, and the number of words to be written.

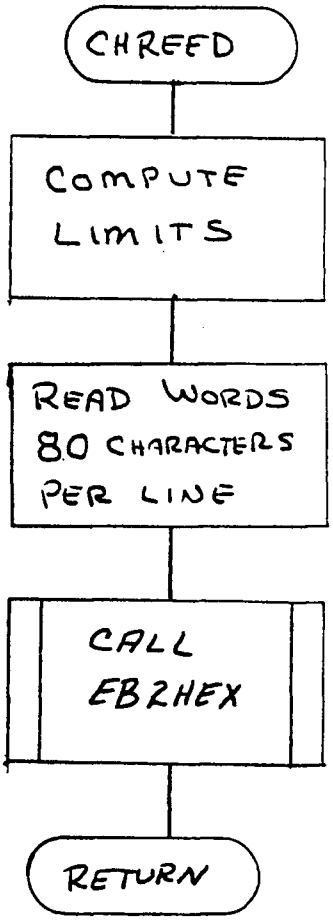
The routine calls HEX2EB to format the characters for printing and then prints them 80 characters to a line.

2.3.6 Subroutine CHREED

The Fortran subroutine CHREED handles a request by the user's program to read in character format. Subroutine FIBM passes it three parameters: the first byte address, the corresponding index in the NW array, and the number of words to be read.

The routine reads the characters from the INPUT file and then calls EB2HEX to convert them to internal hexadecimal format.


CHRITE
COMPUTE LIMITS
CALL HEX2EB
PRINT WORDS 80 CHARACTERS PER LINE
RETURN


CHREED
COMPUTE LIMITS
READ WORDS 80 CHARACTERS PER LINE
CALL EB2HEX
RETURN

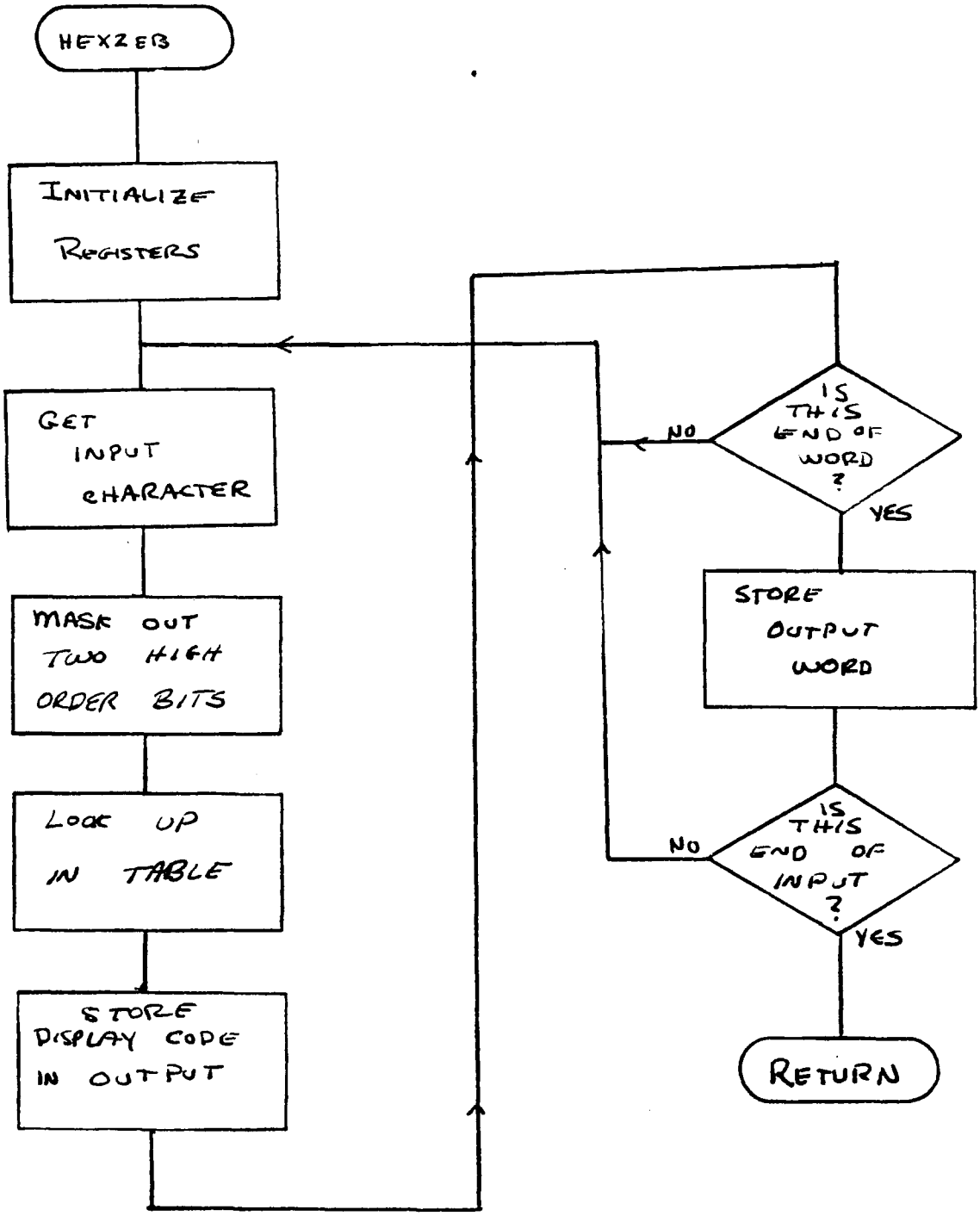
2.3.7 Subroutine EBCDIC

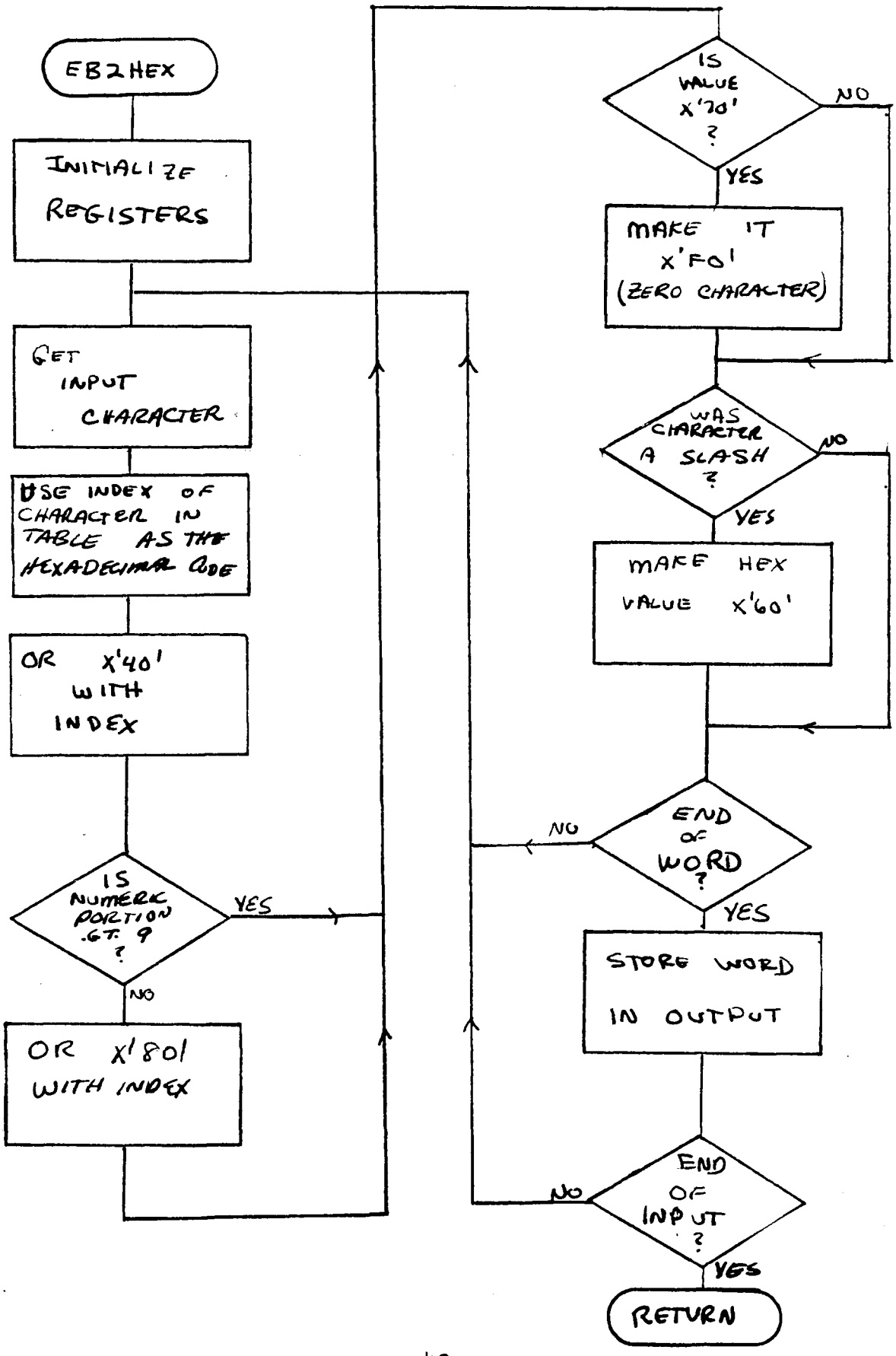
EBCDIC is a COMPASS subroutine consisting of two distinct routines. The two routines, HEX2EB and EB2HEX, convert words between the internal hexadecimal format and the corresponding character representation. Both routines are based on a table of SCOPE display code characters arranged in hexadecimal order. The displacement in this table corresponds to the low-order six bits of the hexadecimal byte.

HEX2EB translates single bytes by ignoring the two high-order bits. Using the remaining six bits, it finds the corresponding display code character in the table.

EB2HEX performs the inverse function. It finds a display code character in the table; then, using the index as the low-order six bits, it generates the appropriate bits to fill the eight-bit byte.

Both routines are passed a starting address and the number of words to be converted. The converted words are stored in place of the unconverted words in the array.


HEXZEB
INITIALIZE REGISTERS
GET INPUT CHARACTER
MASK OUT TWO HIGH ORDER BITS
LOOK UP IN TABLE
STORE DISPLAY CODE IN OUTPUT
IS THIS END OF WORD ?
NO
YES
STORE OUTPUT WORD
IS THIS END OF INPUT ?
NO
YES
RETURN


EB2HEX
INIMALIZE REGISTERS
GET INPUT CHARACTER
USE INDEX OF CHARACTER IN TABLE AS THE HEXADECIMAL CODE
OR X'40' WITH INDEX
IS NUMERIC PORTION .GT. 9 ?
YES
NO
OR X'80' WITH INDEX
IS VALUE X'70' ?
NO
YES
MAKE IT X'F0' (ZERO CHARACTER)
WAS CHARACTER A SLASH ?
NO
YES
MAKE HEX VALUE X'60'
END OF WORD ?
NO
YES
STORE WORD IN OUTPUT
END OF INPUT ?
NO
YES
RETURN

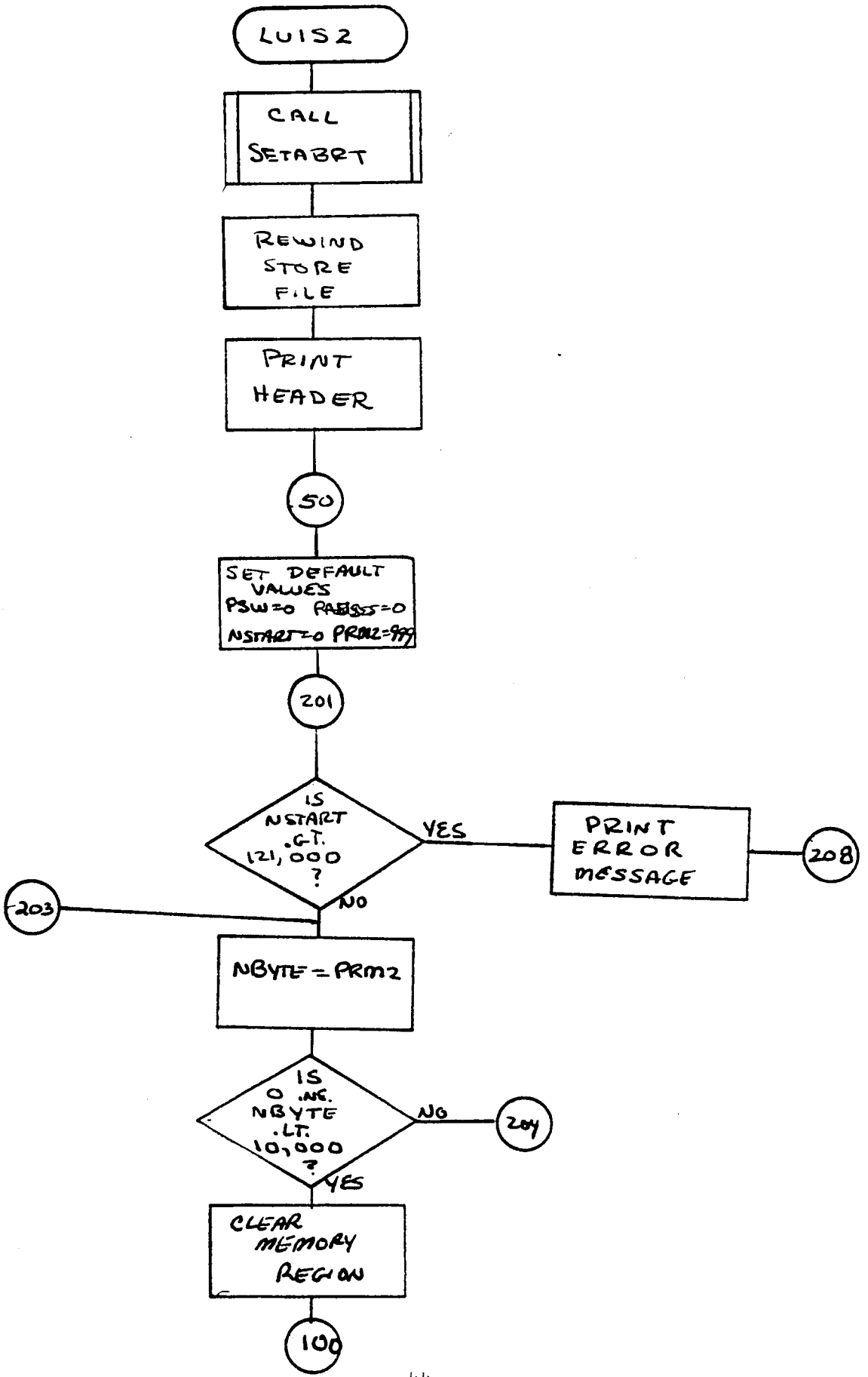
2.3.8 The Main Routine LUIS2

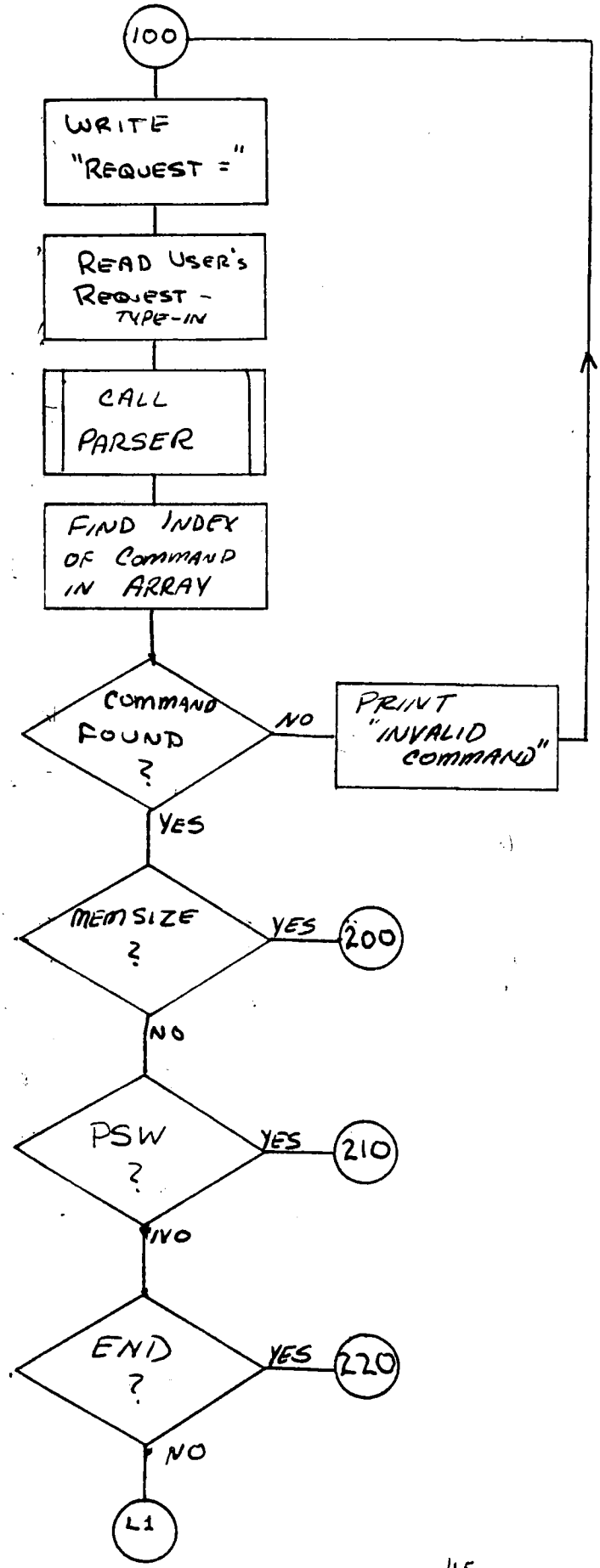
The Fortran routine LUIS2 replaces the routine LUIS of the old simulator. LUIS2 performs all of the functions of LUIS in addition to some new ones. The interfaces to the other simulator routines are in no way changed; LUIS2 simply provides an improved user interface.

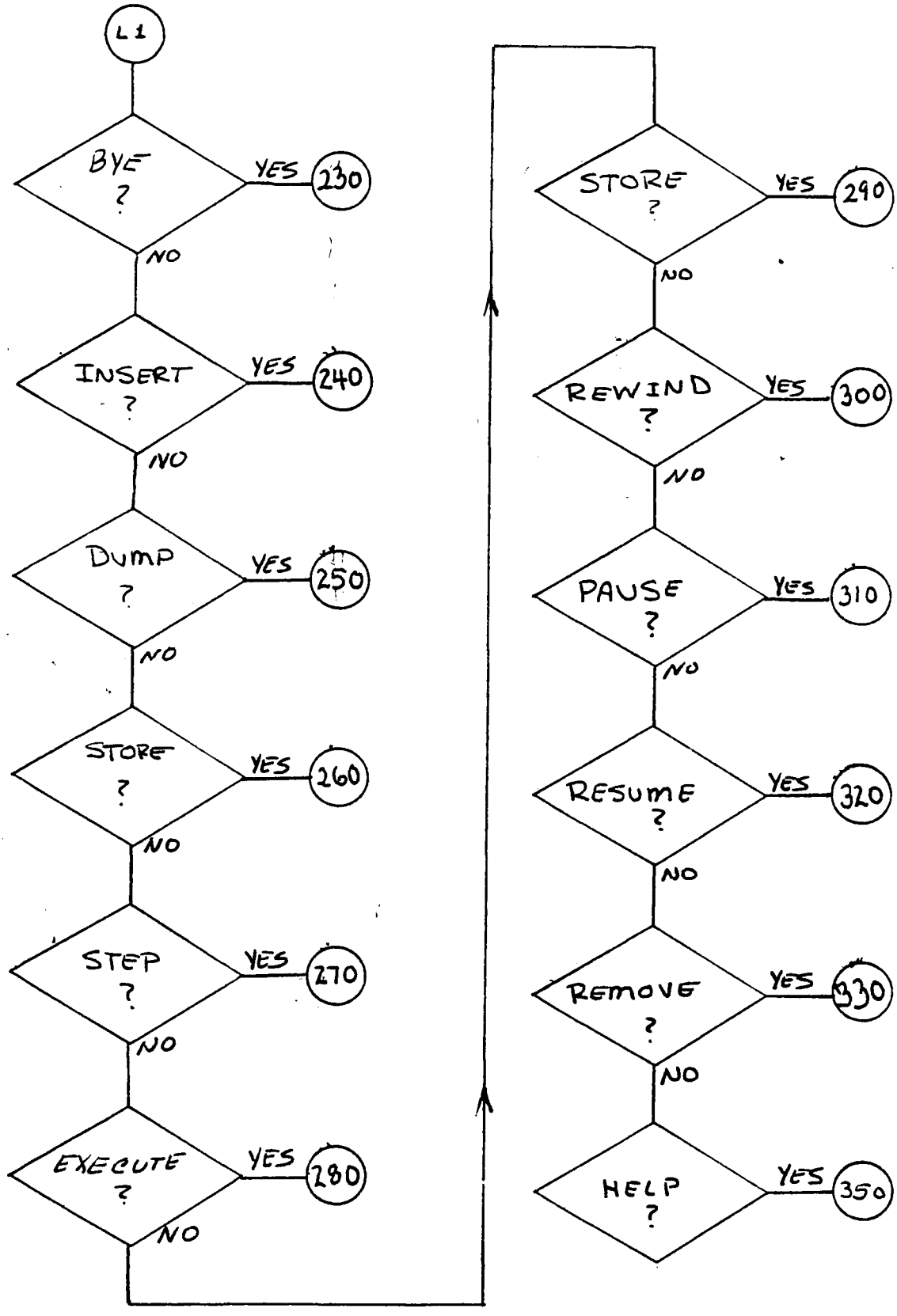
After setting certain default values, LUIS2 immediately puts the user in REQUEST mode. When a request is entered, the subroutine PARSER is called to separate the various parameters of the command. When control is returned, LUIS2 looks up the command word in the command array (CMDARY). If it is not found an error is printed; if it is found the index into the array is used in a COMPUTED GO TO to transfer control to the appropriate code for that command.

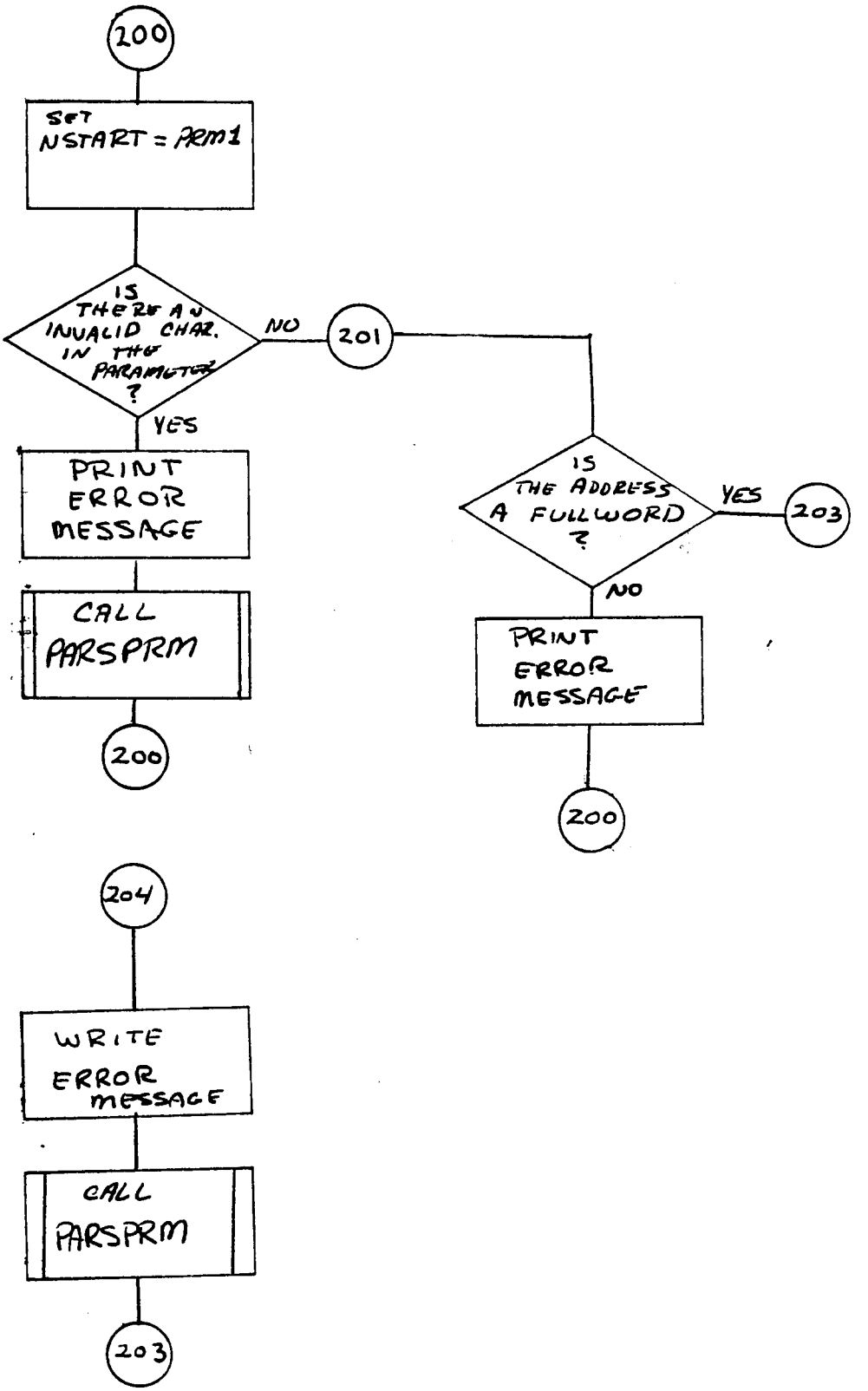
If, at any time, an invalid parameter is detected, an error is printed and PARSPRM is called to read the revised parameter as entered by the user. When finished processing any command (except BYE), control is returned to REQUEST mode.

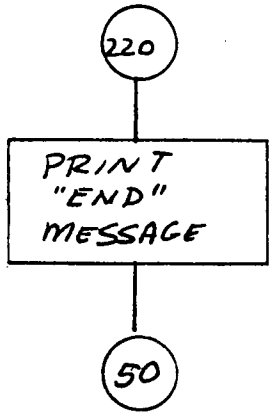
In the flowchart which follows, all numbers found in connectors correspond to the statement numbers of the associated code. Not all statement numbers appear in the flowchart, however. Wherever possible, the flowchart is arranged in the same sequence as the code in the program. In a few cases the order was changed in the interest of clarity.

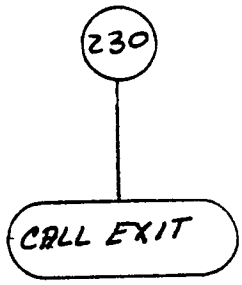

LUIS2
CALL SETABRT
REWIND STORE FILE
PRINT HEADER
50
SET DEFAULT VALUES
PSW=0 PARMS=0
NSTART=0 PRM2=999
201
IS NSTART .GT. 121,000 ?
YES
PRINT ERROR MESSAGE
208
NO
203
NBYTE = PRM2
IS 0 .NE. NBYTE .LT. 10,000 ?
NO
204
YES
CLEAR MEMORY REGION
100

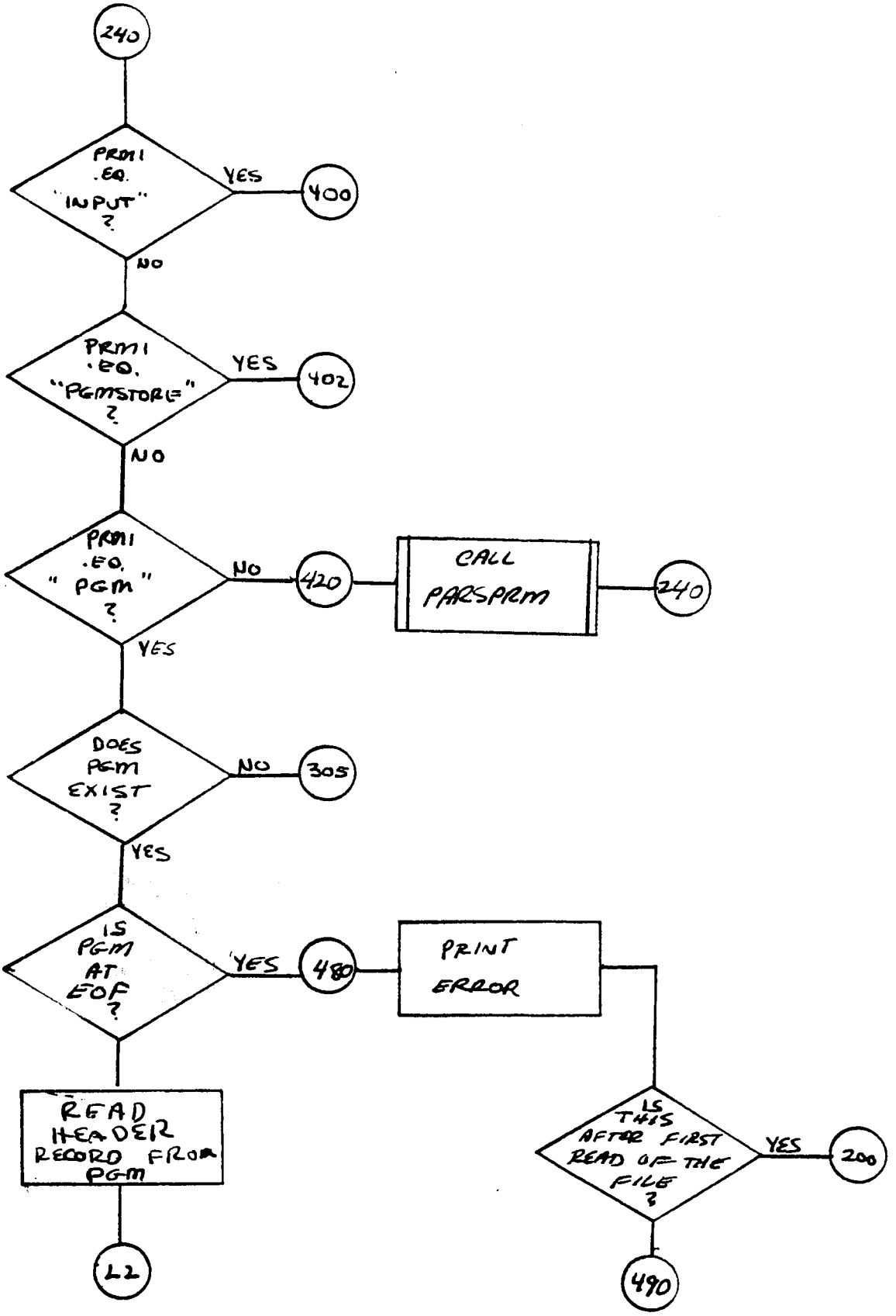

100
WRITE "REQUEST ="
READ USER's REQUEST - TYPE-IN
CALL PARSER
FIND INDEX OF COMMAND IN ARRAY
COMMAND FOUND ?
NO
PRINT "INVALID COMMAND"
YES
MEMSIZE ?
YES
200
NO
PSW ?
YES
210
NO
END ?
YES
220
NO
L1

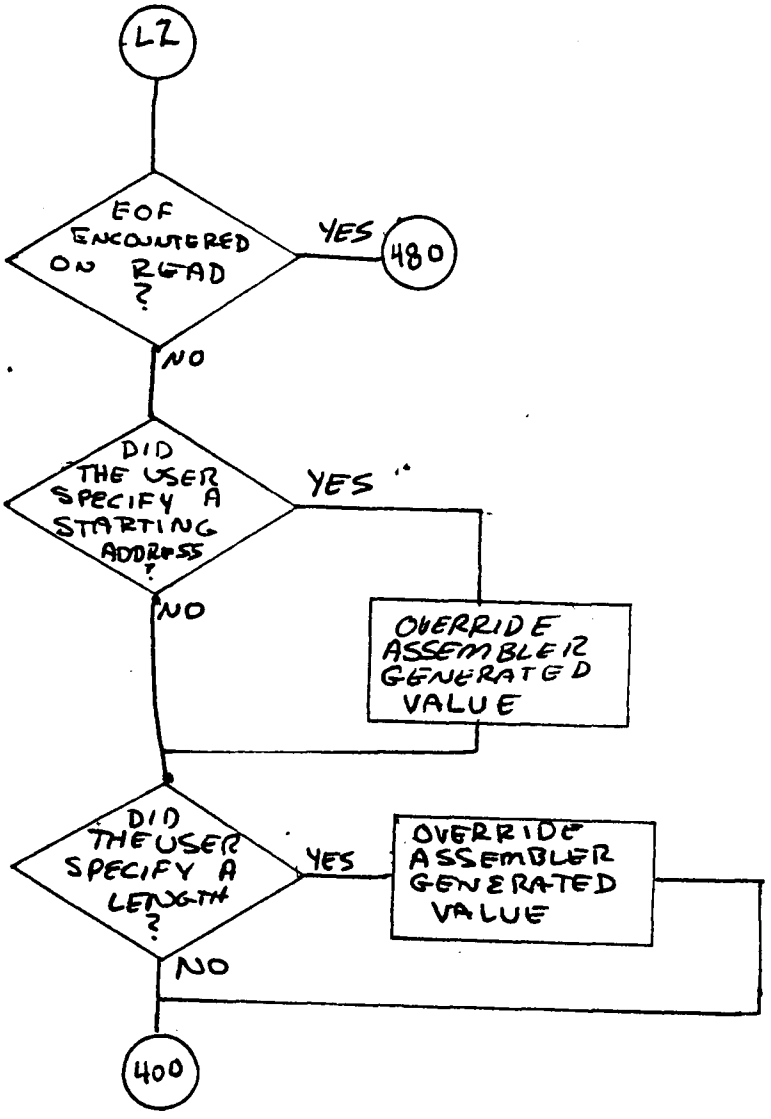

L1
BYE ?
YES
230
NO
INSERT ?
YES
240
NO
DUMP ?
YES
250
NO
STORE ?
YES
260
NO
STEP ?
YES
270
NO
EXECUTE ?
YES
280
NO
STORE ?
YES
290
NO
REWIND ?
YES
300
NO
PAUSE ?
YES
310
NO
RESUME ?
YES
320
NO
REMOVE ?
YES
330
NO
HELP ?
YES
350

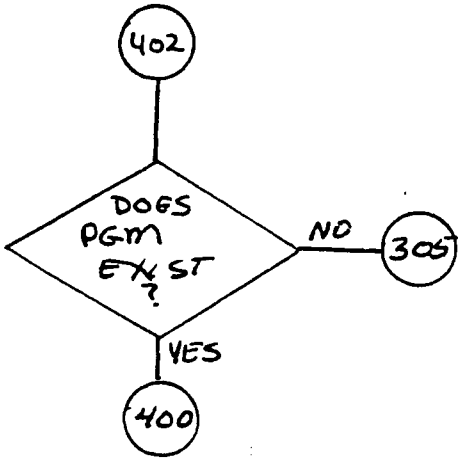

200
SET NSTART = PRM1
IS THERE AN INVALID CHAR. IN THE PARAMETER ?
NO
201
YES
PRINT ERROR MESSAGE
CALL PARSPRM
200
IS THE ADDRESS A FULLWORD ?
YES
203
NO
PRINT ERROR MESSAGE
200
204
WRITE ERROR MESSAGE
CALL PARSPRM
203

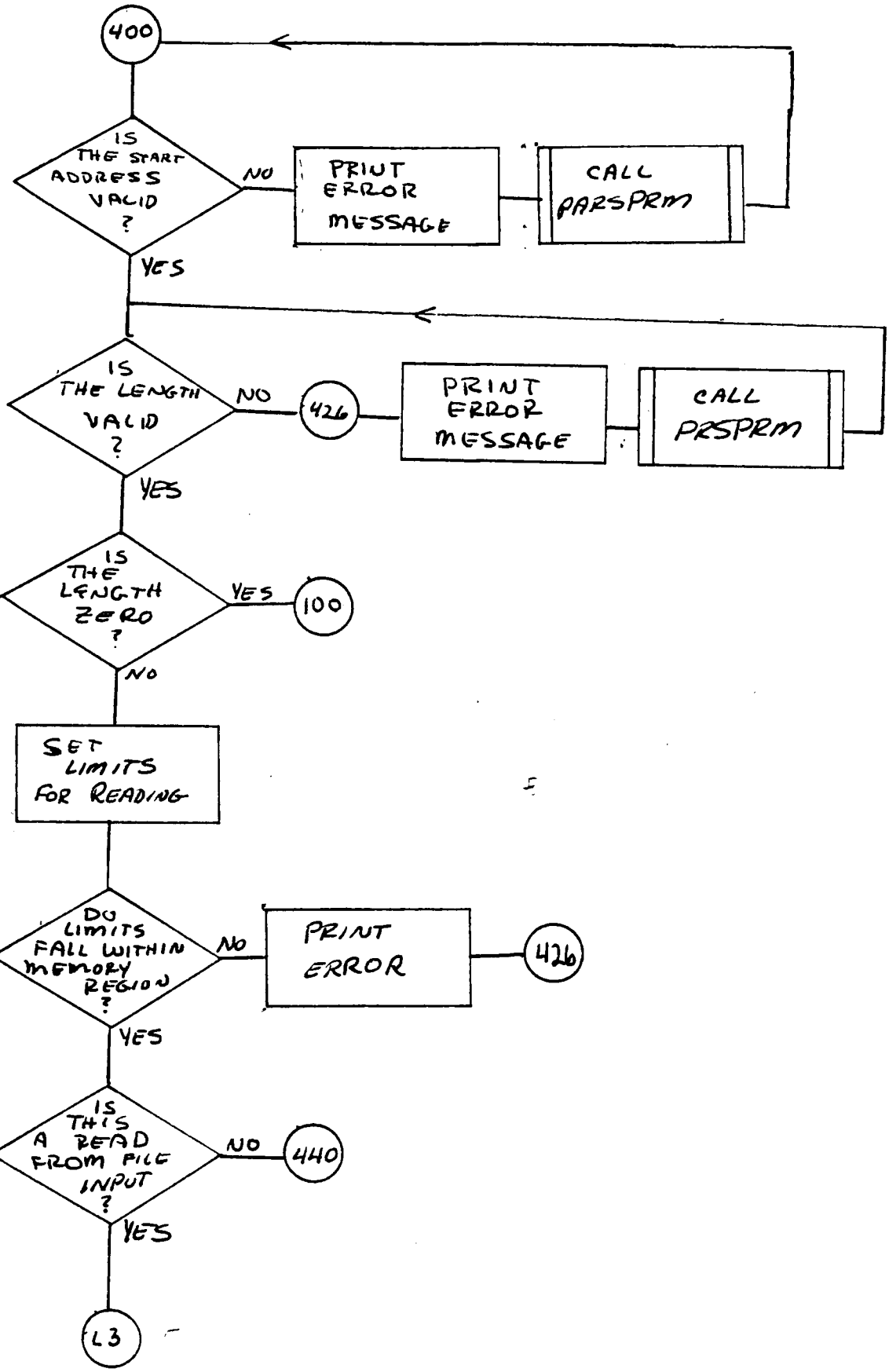

220
PRINT
"END"
MESSAGE
50

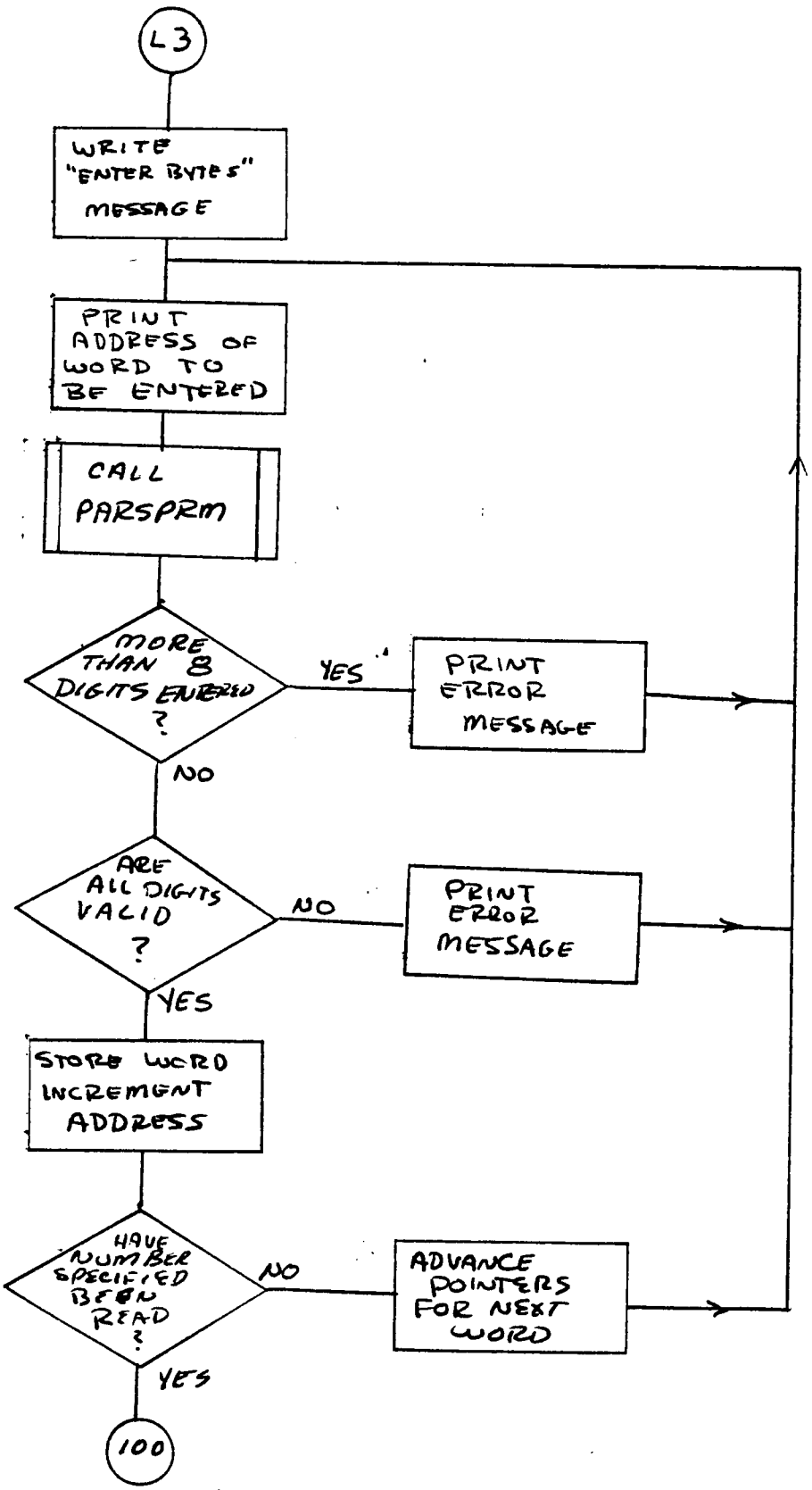

230
CALL EXIT

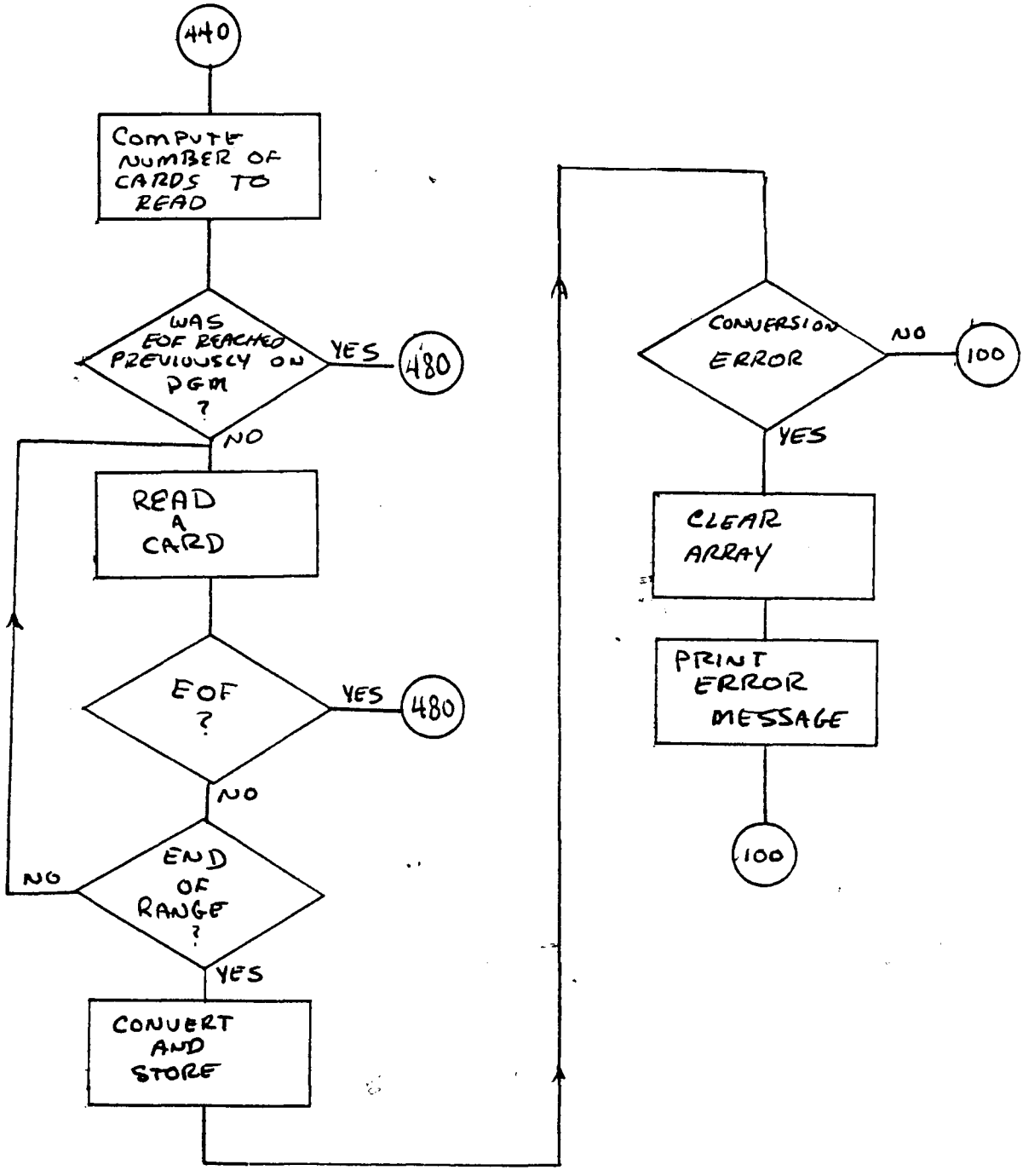

240
PRM1 .EQ. "INPUT" ?
YES
400
NO
PRM1 .EQ. "PGMSTORE" ?
YES
402
NO
PRM1 .EQ. "PGM" ?
NO
420
CALL PARSPRM
240
YES
DOES PGM EXIST ?
NO
305
YES
IS PGM AT EOF ?
YES
480
PRINT ERROR
READ HEADER RECORD FROM PGM
L2
IS THIS AFTER FIRST READ OF THE FILE ?
YES
200
490

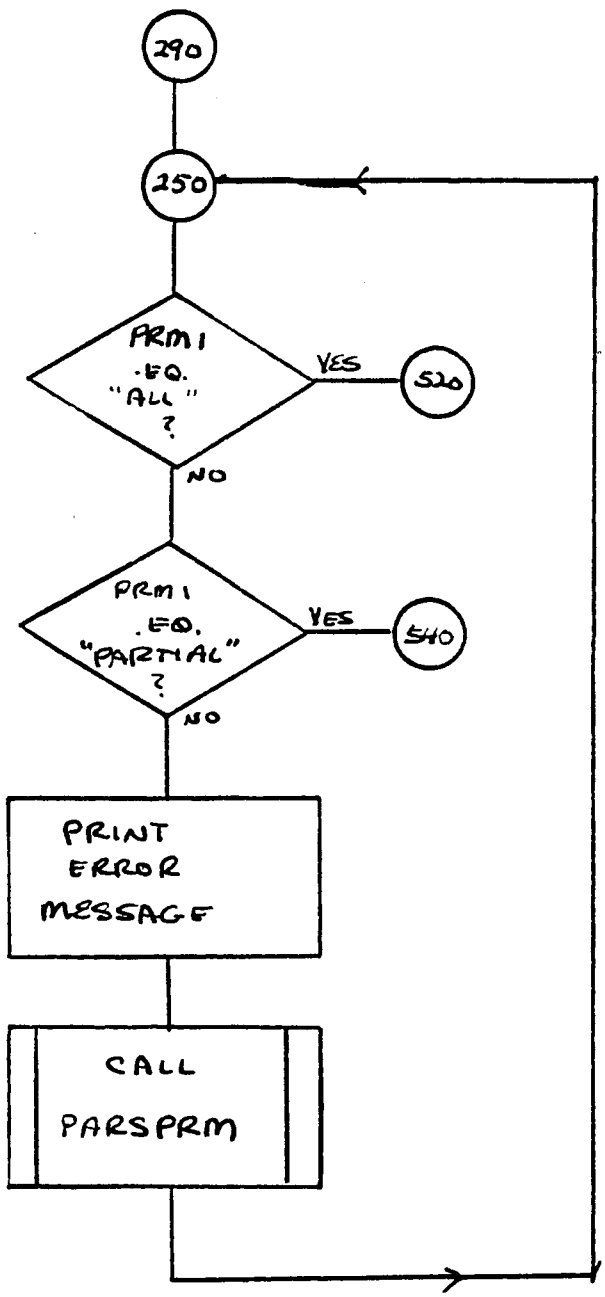

L2
EOF ENCOUNTERED ON READ ?
YES
480
NO
DID THE USER SPECIFY A STARTING ADDRESS ?
YES
NO
OVERRIDE ASSEMBLER GENERATED VALUE
DID THE USER SPECIFY A LENGTH ?
YES
OVERRIDE ASSEMBLER GENERATED VALUE
NO
400

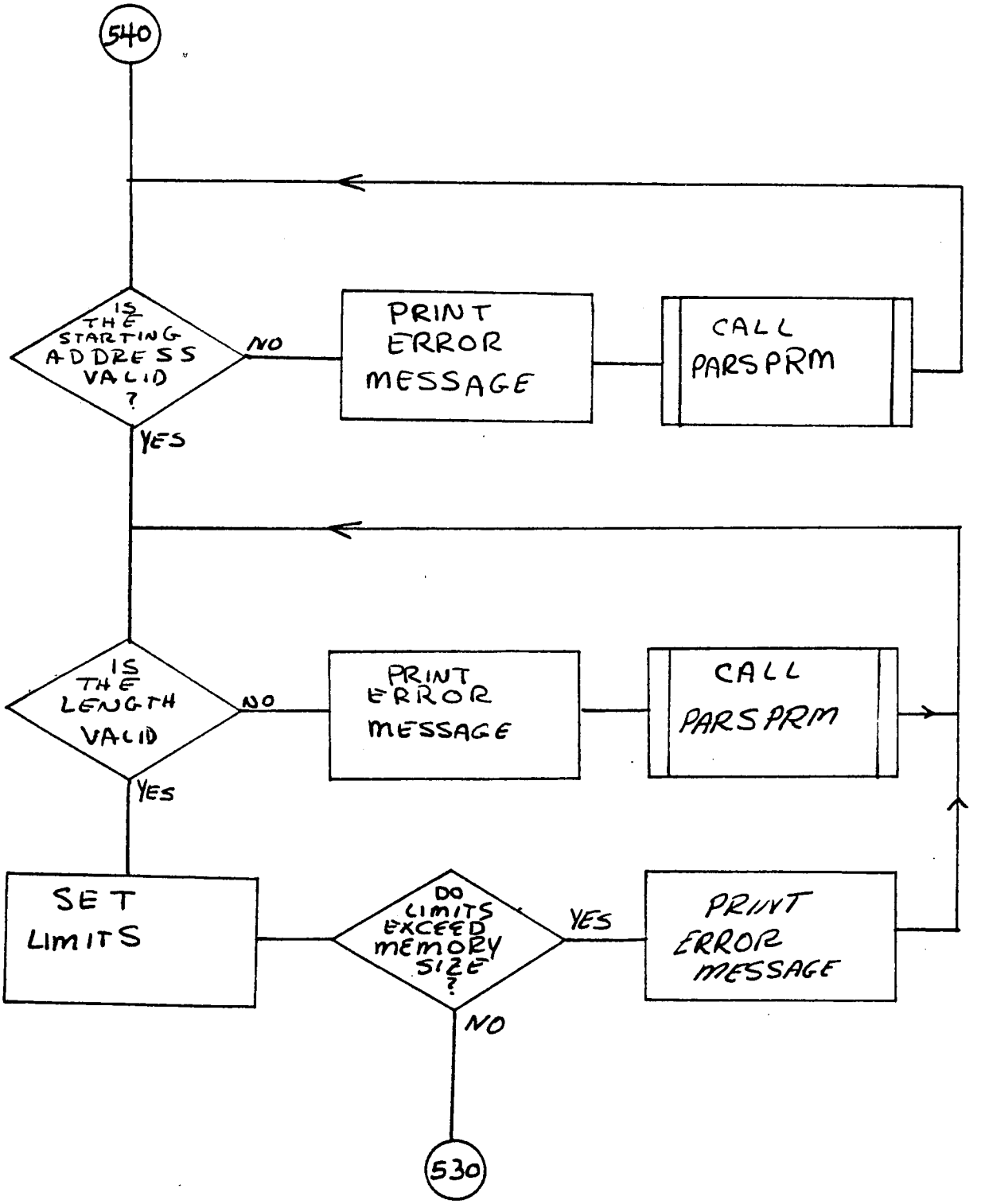

402
DOES PGM EXIST ?
NO
305
YES
400

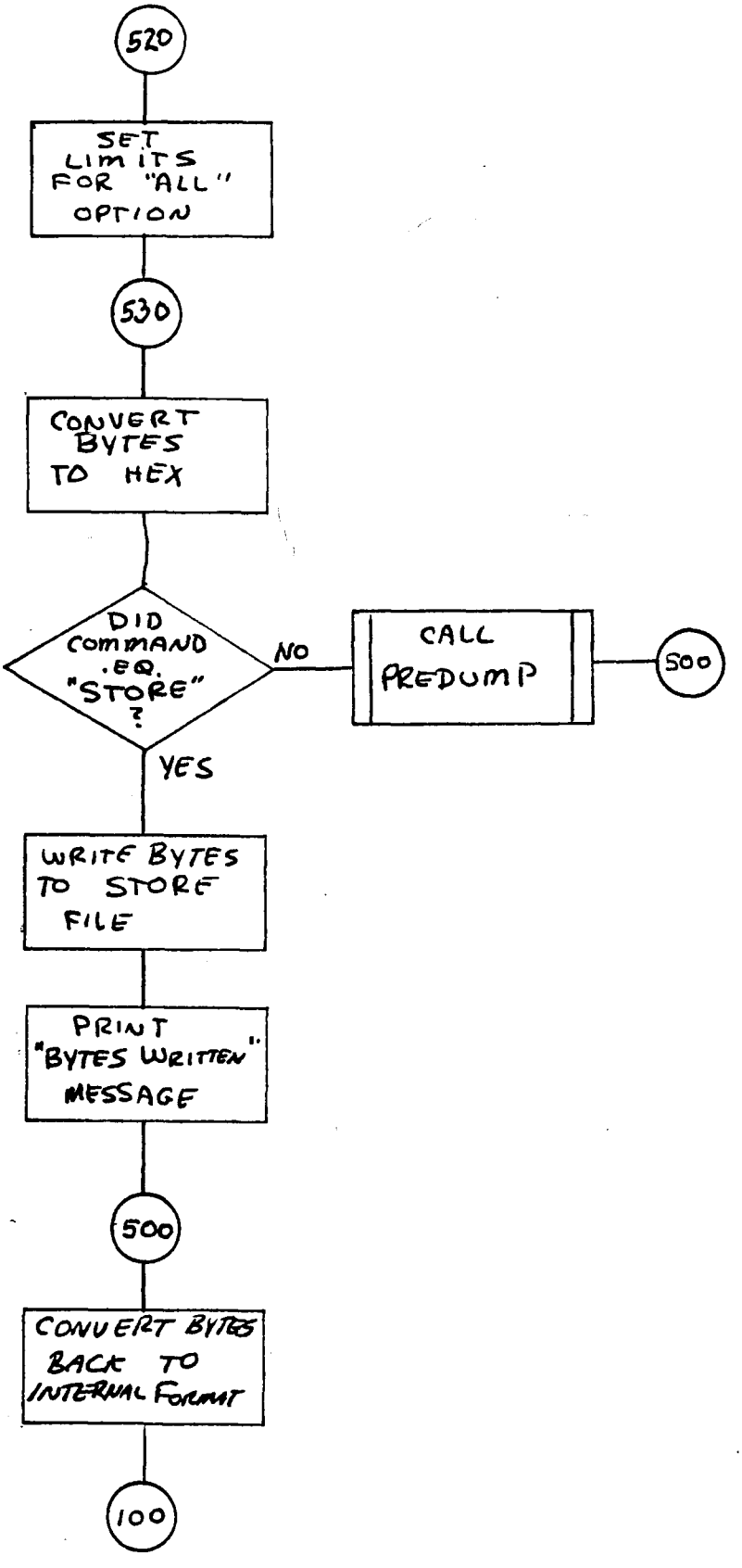

400
IS THE START ADDRESS VALID ?
NO
PRINT ERROR MESSAGE
CALL PARSPRM
YES
IS THE LENGTH VALID ?
NO
426
PRINT ERROR MESSAGE
CALL PRSPRM
YES
IS THE LENGTH ZERO ?
YES
100
NO
SET LIMITS FOR READING
DO LIMITS FALL WITHIN MEMORY REGION ?
NO
PRINT ERROR
426
YES
IS THIS A READ FROM FILE INPUT ?
NO
440
YES
L3

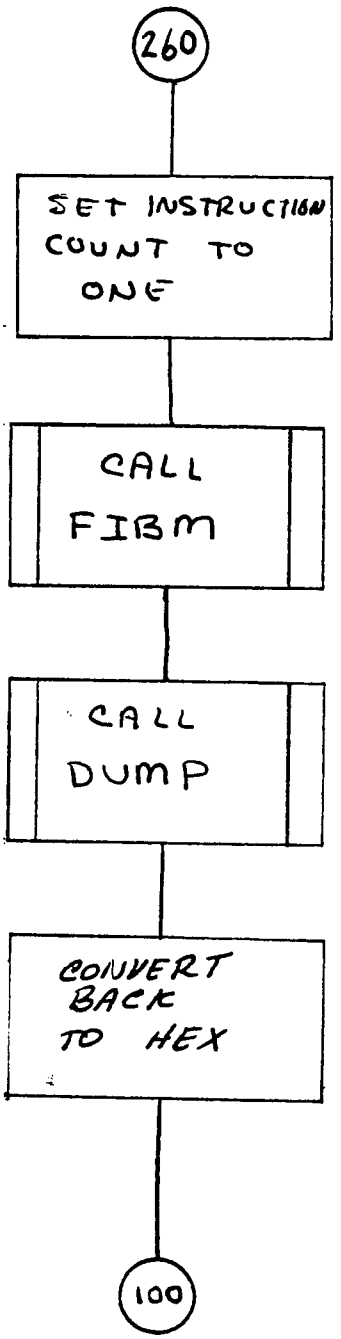

L3
WRITE "ENTER BYTES" MESSAGE
PRINT ADDRESS OF WORD TO BE ENTERED
CALL PARSPRM
MORE THAN 8 DIGITS ENTERED ?
YES
PRINT ERROR MESSAGE
NO
ARE ALL DIGITS VALID ?
NO
PRINT ERROR MESSAGE
YES
STORE WORD INCREMENT ADDRESS
HAVE NUMBER SPECIFIED BEEN READ ?
NO
ADVANCE POINTERS FOR NEXT WORD
YES
100

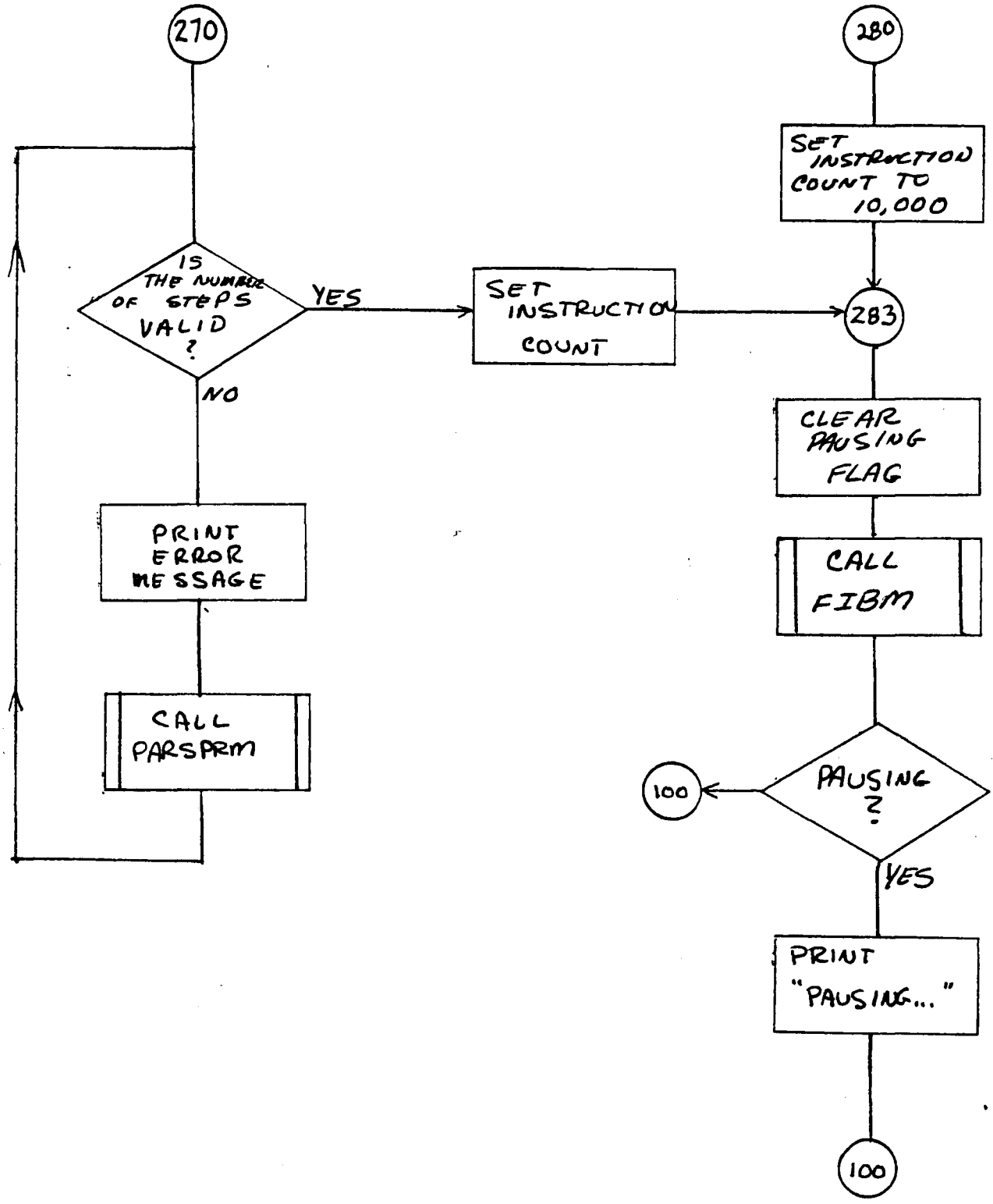

440
COMPUTE NUMBER OF CARDS TO READ
WAS EOF REACHED PREVIOUSLY ON PGM ?
YES
480
NO
READ A CARD
EOF ?
YES
480
NO
END OF RANGE ?
NO
YES
CONVERT AND STORE
CONVERSION ERROR
NO
100
YES
CLEAR ARRAY
PRINT ERROR MESSAGE
100

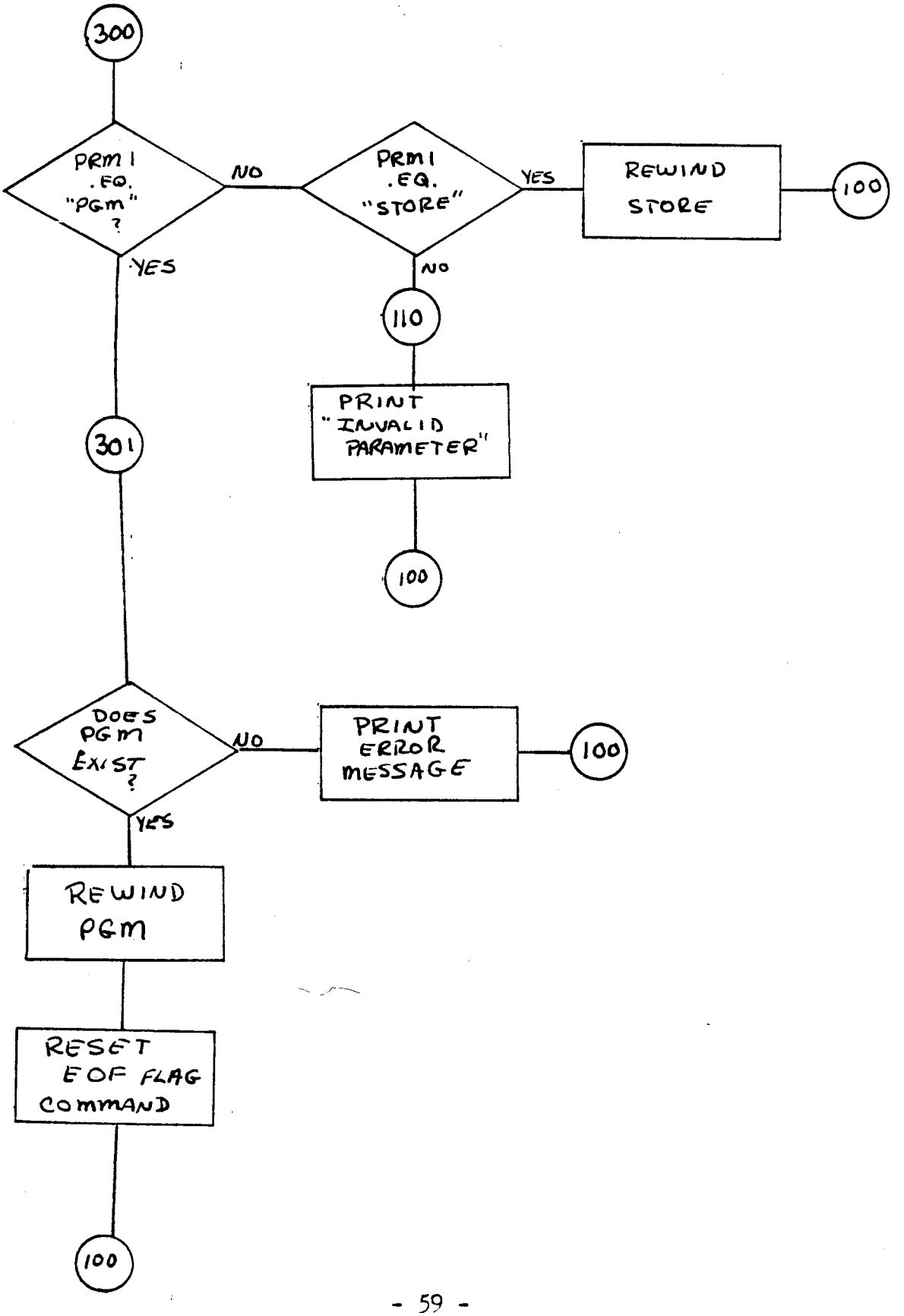

290
250
PRM1
.EQ.
"ALL"
?
YES
520
NO
PRM1
.EQ.
"PARTIAL"
?
YES
540
NO
PRINT
ERROR
MESSAGE
CALL
PARSPRM

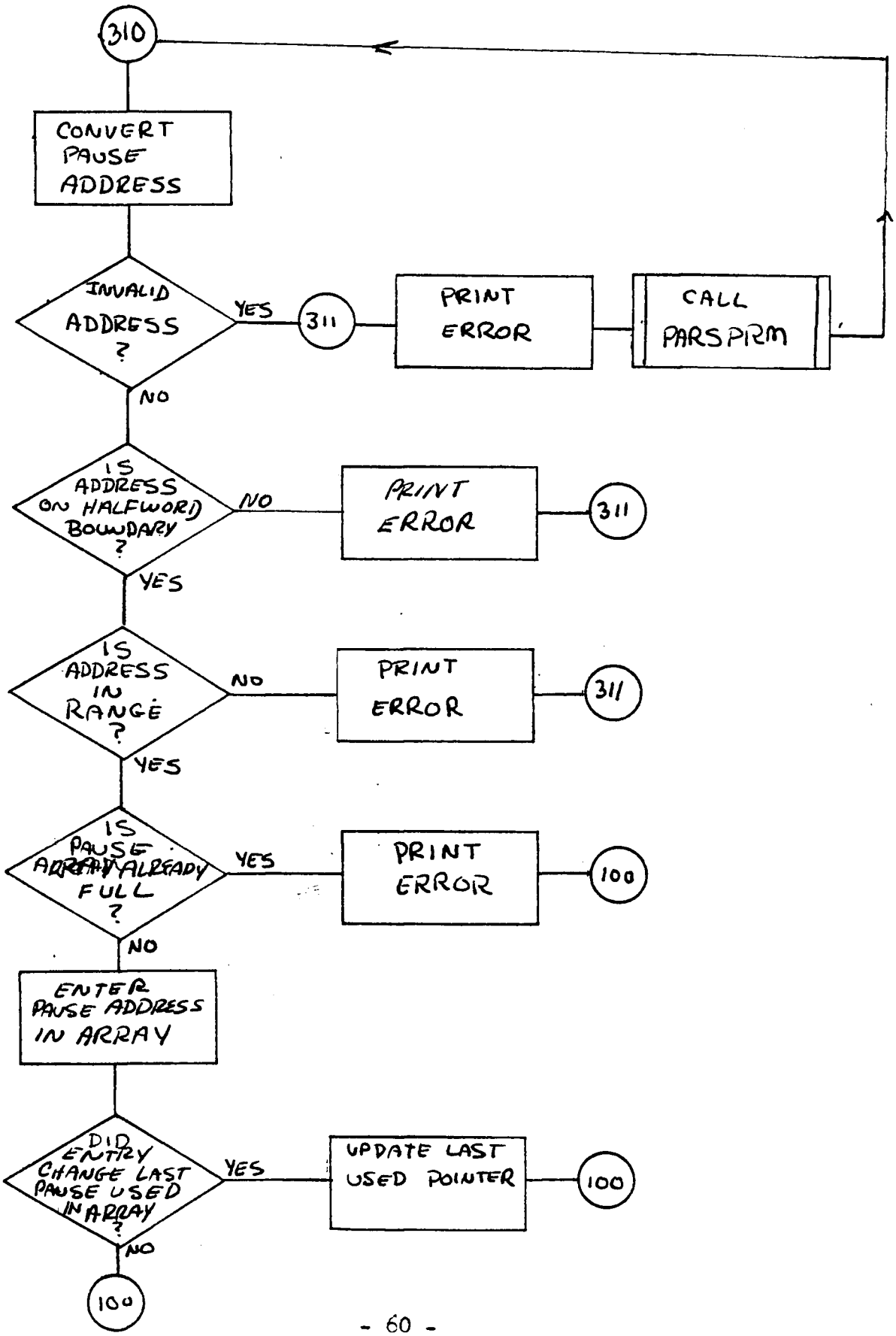

540
IS THE STARTING ADDRESS VALID ?
NO
PRINT ERROR MESSAGE
CALL PARSPRM
YES
IS THE LENGTH VALID
NO
PRINT ERROR MESSAGE
CALL PARSPRM
YES
SET LIMITS
DO LIMITS EXCEED MEMORY SIZE ?
YES
PRINT ERROR MESSAGE
NO
530

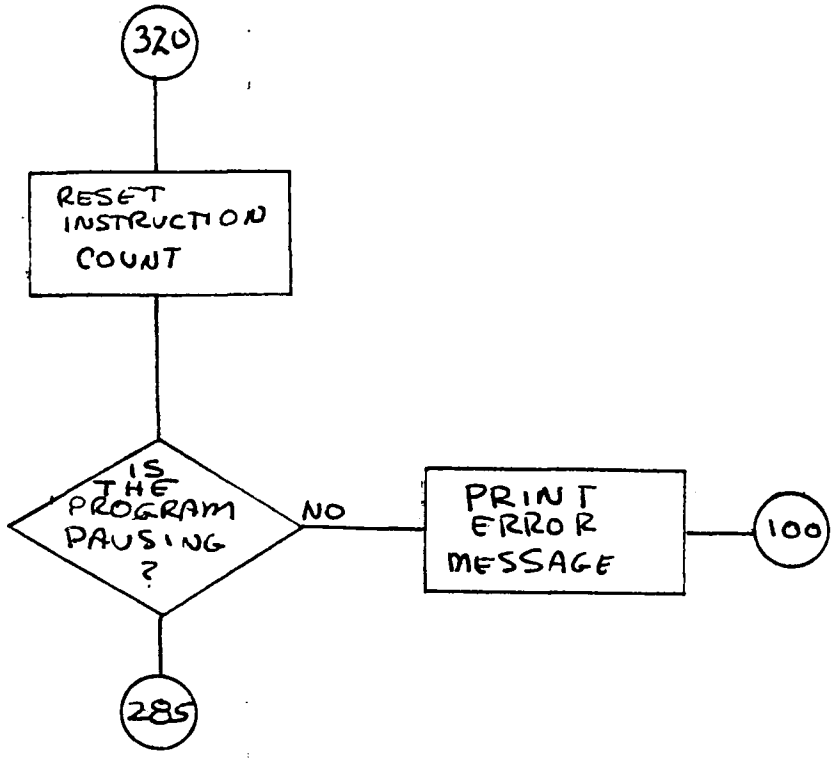

520
SET LIMITS FOR "ALL" OPTION
530
CONVERT BYTES TO HEX
DID COMMAND .EQ. "STORE" ?
NO
CALL PREDUMP
500
YES
WRITE BYTES TO STORE FILE
PRINT "BYTES WRITTEN" MESSAGE
500
CONVERT BYTES BACK TO INTERNAL FORMAT
100

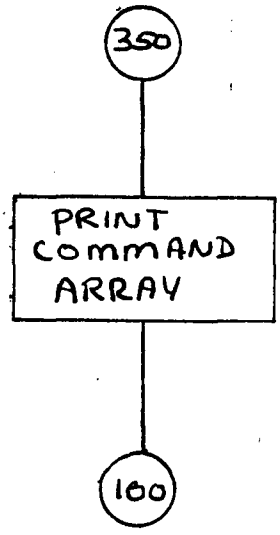

260
SET INSTRUCTION COUNT TO ONE
CALL FIBM
CALL DUMP
CONVERT BACK TO HEX
100

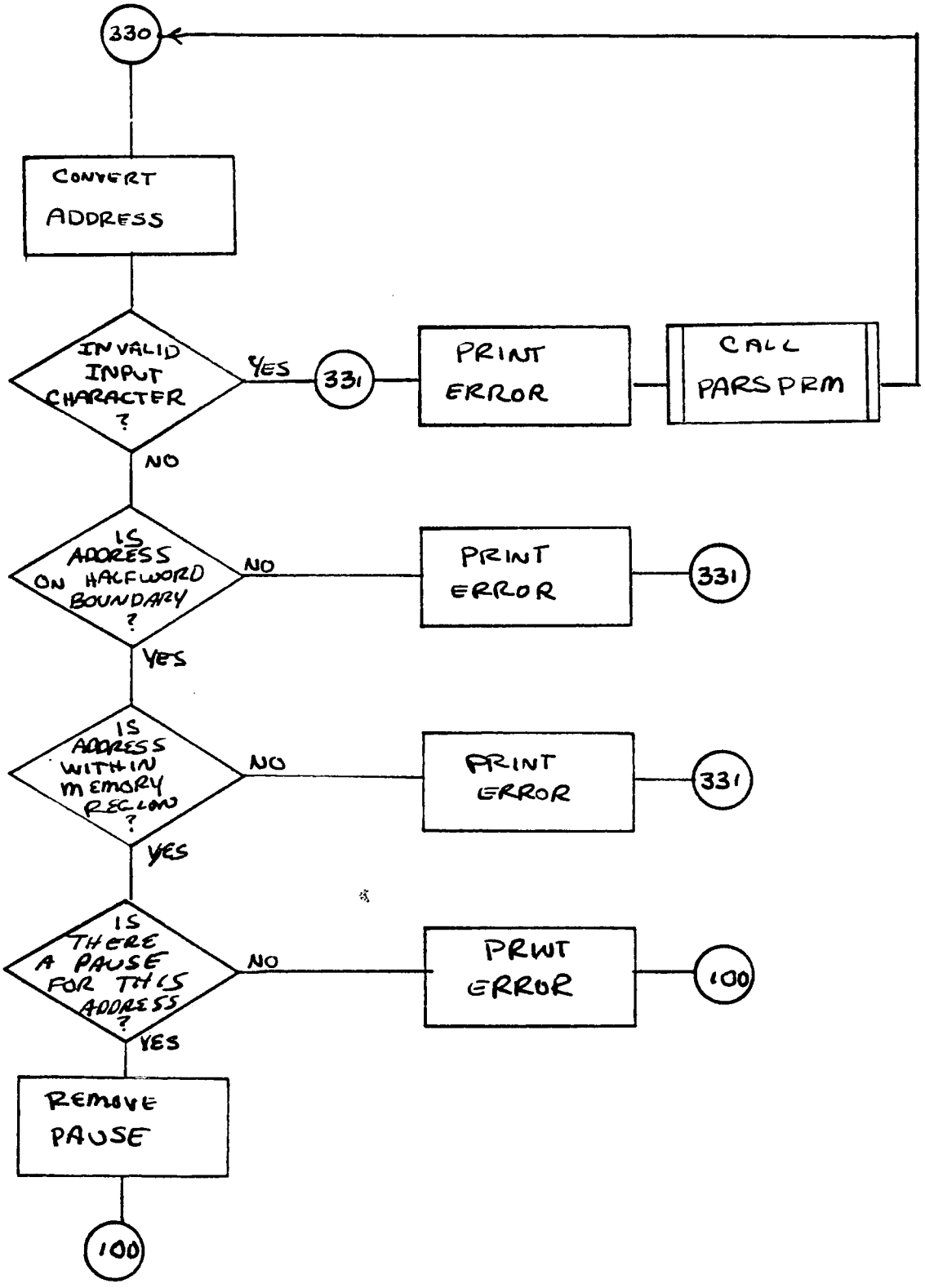

270
IS THE NUMBER OF STEPS VALID ?
YES
NO
SET INSTRUCTION COUNT
PRINT ERROR MESSAGE
CALL PARSPRM
280
SET INSTRUCTION COUNT TO 10,000
283
CLEAR PAUSING FLAG
CALL FIBM
PAUSING ?
100
YES
PRINT "PAUSING..."
100


300
PRM1 .EQ. "PGM" ?
NO
PRM1 .EQ. "STORE"
YES
REWIND STORE
100
YES
NO
110
PRINT "INVALID PARAMETER"
100
301
DOES PGM EXIST ?
NO
PRINT ERROR MESSAGE
100
YES
REWIND PGM
RESET EOF FLAG COMMAND
100


310
CONVERT PAUSE ADDRESS
INVALID ADDRESS ?
YES
311
PRINT ERROR
CALL PARSPRM
NO
IS ADDRESS ON HALFWORD BOUNDARY ?
NO
PRINT ERROR
311
YES
IS ADDRESS IN RANGE ?
NO
PRINT ERROR
311
YES
IS PAUSE ARRAY ALREADY FULL ?
YES
PRINT ERROR
100
NO
ENTER PAUSE ADDRESS IN ARRAY
DID ENTRY CHANGE LAST PAUSE USED IN ARRAY ?
YES
UPDATE LAST USED POINTER
100
NO
100


320
RESET INSTRUCTION COUNT
IS THE PROGRAM PAUSING ?
NO
PRINT ERROR MESSAGE
100
285


350
PRINT COMMAND ARRAY
100


330
CONVERT ADDRESS
INVALID INPUT CHARACTER ?
YES
331
PRINT ERROR
CALL PARSPRM
NO
IS ADDRESS ON HALFWORD BOUNDARY ?
NO
PRINT ERROR
331
YES
IS ADDRESS WITHIN MEMORY REGION ?
NO
PRINT ERROR
331
YES
IS THERE A PAUSE FOR THIS ADDRESS ?
NO
PRINT ERROR
100
YES
REMOVE PAUSE
100

3.0 The Lehigh University IBM 360 Assembler

This section describes the modifications made to the Lehigh University IBM 360 Assembler (hereafter referred to as LUIAS or the assembler). The current version of the assembler is the result of the combination of two existing programs: one, an existing assembler, and the other a macro processor.

Section 3.1 is the user's manual for the assembler. It supplies the needed information for a user of the assembler. It describes the pseudo-operations available and the major differences between this assembler and an IBM 360 assembler.

Section 3.2 gives a detailed description of the differences between LUIAS and an IBM assembler. It also gives a brief discussion of options available to a user to circumvent certain of the limitations of LUIAS.

Section 3.3 describes the motivations behind the modifications to the assembler. It also explains some of the problems encountered in attempting to interface the two existing programs.

Section 3.4 provides a detailed description of the changes made and the routines added to the assembler. Included are detatiled flowcharts showing the logic flow within the added subroutines.

Section 3.5 describes a number of changes which might be made to LUIAS at a later date.

3.1 User's Guide to the Lehigh University IBM Assembler

This section is a reproduction of the user's manual which will be made available to the users of this assembler. Because it is intended for the user, it contains no references to the code or the internal routines of LUIAS. Instead, it contains a description of the assembler pseudo-ops and instruction formats which a user of the assembler must know. It is assumed that the reader of this user's guide is familiar with IBM 360 assembly language, and thus special emphasis is placed on the aspects of LUIAS which differ from IBM.

It should be noted that much of the material contained in this user's manual is adapted from similar manuals prepared by Maroudas (4) and Seager (5) for the macro generator and an earlier version of the assembler.

The Lehigh University IBM Assembler User Guide

Introduction

The Lehigh Univesity IBM Assembler (LUIAS) converts a program written in IBM System/360 assembly language into IBM System/360 machine language. The output of the assembler includes source and object listings and the assembled code in a format compatible with the Lehigh University IBM 360 Simulator (LUIS). The assembler processes three types of statements: source statements and associated pseudo-operations, conditional assembly pseudo-operations, and macro call statements.

Program Listings

The assembly listing produced by LUIAS includes all source lines encountered in the input in addition to lines from the expansion of macro calls. With each line of source code is printed the current location counter value, the generated object code (if any), an error code, the source card number, and the macro call level. At the end of the listing is printed the number of errors encountered and the definition of each error which occurred in the program. Also printed are the program starting address, the next available address, and the program length. A list of the actual generated object code is followed

by the section table, the symbol table, the literal table, and the base table.

In addition to the printed output, a message also appears in the job's dayfile indicating the total number of errors encountered and the number of CPU seconds required for the assembly.

Abort Conditions

Under certain conditions the assembler will abort during the first pass, in which case no source listing will be printed. A single message will be printed indicating one of the following error conditions:

1. No END card encountered.
2. More than 10,000 bytes of code generated.
3. More than 100 symbols defined.
4. More than 50 literals defined.
5. More than 8 control sections or 8 dummy sections defined.
6. No MEND card for a macro definition.
7. A source level sequence symbol not defined after a reference to it occurs.
8. Macro call stack overflow. (more than 1000 entries)
9. More than 100 macro definitions.
10. More than 1000 lines of macro definitions.
11. More than 1000 dummy arguments and SET symbols defined.

Input Format

All input to the assembler may be entered in free format. Labels must start in column one and contain at most five characters. The first character must be alphabetic (A-Z); the remaining ones may be alphanumeric (A-Z,0-9). Any characters beyond the first five are ignored. At least one blank must separate operators from labels and similarly operands from operators. A comment may appear on any source level statement, providing it is separated from the operand by at least one blank. Comments are not permitted on macro prototype statements or LCL and GBL pseudo-operations.

Instruction Set

All but seven instructions of the IBM System/360 standard instruction set are assembled. Omitted are Diagnose, Set System Mask, Load PSW, Halt I/O, Supervisor Call, Test Channel, and Test I/O. The Start I/O (SIO) instruction is modified for compatibility with the simulator. (For a description of the SIO parameters, refer to the LUIS user guide.) In addition to the standard instructions, 18 extended mnemonics and 25 assembler directives are processed. A complete list of all mnemonics is given in Appendix B.

Operands

The maximum implied length for any label in an operand is six bytes; larger lengths must be specified explicitly when required in SS instructions.

Expressions in operands may be either relocatable or absolute. Any number of labels, constants, and location counter references may appear in an expression with the operators plus (+) and minus(-). Relocatibility of an expression is determined as the expression is processed from left to right, making it possible for one arrangement of terms to be illegal while an algebraicly equivalent rearrangement is valid. Stated simply the rules governing the formation of expressions are:

1. Two relocatable terms may not be added.

2. The difference of two relocatable terms is always absolute.

3. An absolute term may be added to or subtracted from any other term.

4. Two relative terms can be subtracted only when both are in the same CSECT or DSECT.

Any relative address which appears in an instruction operand must be covered by a base register. Base registers must be assigned for each CSECT and DSECT that is to be addressed. A single base register cannot simultaneously cover more than one section.

Pseudo-Operations

The pseudo-operations processed by the assembler fall into three categories: source level, conditional assembly, and macro processing.

There are 13 source level pseudo-ops which control the

location counter, define labels and base register assignments, and define data constants.

1. START defines the starting address and name of the program. The operand may be any fixed point number or blank.
2. ORG controls the position of the location counter. The operand may be any relocatable expression in the current control section. When the location counter is repositioned, intervening bytes are left unchanged. Bytes may be lost if the location counter is moved backward and not returned to its largest previous value before the control section or program ends.
3. USING defines base register usage. Any number of registers may be specified.
4. DROP removes register from list of available base registers. Only one register may be specified.
5. LTORG advances the location counter to a double word boundary and causes literals referenced since the START card or last LTORG to be stored. The CSECT in which a literal is referenced has no effect upon where it is stored; the user must assure that literals are properly covered.
6. END indicates the physical end of the program. Any literals not previously stored are stored at the end of the current CSECT.

7. CSECT the label field indicates the name of the control section to be started or resumed.

8. DSECT the label field indicates the name of the dummy section to be started or resumed. Any instruction or pseudo-instrucion may appear in a DSECT but no code will be generated. Locations within a DSECT may not be addressed by an address constant.

9. CNOP conditionally generates NOPR instrucions to advance the location counter to the specified halfword boundary. Valid operands are:

   0,4 2,4 0,8 2,8 4,8 6,8

   The second digit indicates fullword (4) or double word (8) alignment; the first digit, the number of bytes to advance past that boundary.

10. EQU defines the label to have the value and attributes of the expression given in the operand.

11. DS any of the following constant types may be specified: A, B, C, F, H, P, X, Z. The optional fixed point multiplier may be any positive integer or zero. Data type D may only appear with a zero length. The explicit length parameter is not permitted with any data type.

12. DC Any of the above mentioned data types, except D, may be specified with a duplication factor.

All data types use parentheses as delimiters; the apostrophes are not used. Multiple operands may be specified separated by commas. Only a single constant may appear when a duplication factor is specified. For example,

```
DC    F(1,2,36)
DC    23F(126)
```

A length may not be specified for any data constant. No constant appearing in a literal or DC statment may be more than six characters in length and the data generated by the DC statement is limited to five bytes. The only exception to this is the DC A statement. Its data field may consist of any valid expression. Character constants (type C) are terminated by either a comma or a right parenthesis. These two characters (comma and right parenthesis) can only be generated when they appear as the first character in the character string.

13. IO This pseudo-op is unique to LUIAS. It allows the user to easily define the parameters required by the simulator for doing I/O. Its format is as follows:

```
LABEL   IO   N,IOAR
LABEL   IO   N,B1(D1)
```

N is a fixed point number indicating the number of words and the address points to the area to be transfered by the I/O operation.

The ten conditional assembly pseudo-ops allow the user to control the sequence in which source statments are assembled and to define SET symbol values.

Local SET symbols can only be referenced in the level in which they are defined. Other levels may have local variables with the same names. Global SET symbols may be referenced throughout the entire assembly and in any macro definition, providing that they are declared in each level which references them.

Arithmetic SET symbols may take on any integer value which can be expressed in nine digits including the sign (i.e. $+10^9-1$ through -10^8+1). Boolean SET symbols may take only the values TRUE (1) and FALSE (0). All SET symbols are distinguished by a dollar sign ($) as their first character.

Sequence symbols are used to control the sequence in which source lines are processed. They appear in the label field of any statement except MEND, and are distinguished by a period (.) as their first character.

1. LCLA declares a list of SET symbols to be local and arithmetic.

2. LCLB declares a list of SET symbols to be local and Boolean.

3. GBLA declares a list of SET symbols to be global and arithmetic.

4. GBLB declares a list of SET symbols to be global and Boolean.

The above four pseudo-ops must appear before any reference to the SET symbols they define and also before any macro definition or macro call on that level.

5. SETA assigns an arithmetic value to the SET symbol in the label field. The operand may be an integer constant or an arithmetic expression enclosed in parentheses. At most one of the four operators (+, -, *, and /) may appear in a single SETA statement. For complicated functions, several consecutive SETA statements may be used. A unary minus sign may be specified for any symbol appearing in the expression. Division by zero yields a zero result. Constants may not appear in arithmetic expressions. The following are examples of valid SETA statements:

```
$C      SETA    -2
$A      SETA    10
$B      SETA    (-$A+$C)
$D      SETA    ($B/-$C)
```

In the above example $B will be assigned the value -12 and $D the value -6.

6. SETB assigns a Boolean value to the SET symbol in the label field. The operand may be a constant one (true) or zero (false), or a Boolean expression enclosed in parentheses. One of six relational operators (LT, LE, GT, GE, NE, EQ) or one of three logical operators (AND, OR, XOR) may be specified. If a logical operator is used, each term must have a Boolean value and may be preceded by the logical operator NOT. Spaces must be used to separate operators from SET symbols. The longer of two character strings is considered largest when two are compared. If the two are the same length then the display code values are compared. (In the CDC SCOPE character set, a blank is larger than any alphabetic or numeric character.) If a character string is compared to a numeric value the character string is always considered greatest. A string is considered numeric if and only if it contains all numeric characters and possibly a leading sign (+ or -). The following are examples of valid SETB statements:

```
$X      SETB   1
```

```
$Y    SETB  ($D GT $B)
$Z    SETB  ($X AND NOT $Y)
```

If $B and $D are assigned values as in the SETA staement examples, $Y and $Z will have the values true (1) and false (0) respectively.

7. AGO causes an unconditional skip in the sequence in which statements are processed by the assembler. Its operand contains a sequence symbol (including the period) of the next statement to be processed. In a macro, this statement can be located either forward or backward from the location of the AGO. At the source level, this statement must be in the forward direction.

8. AIF This statement causes a skip in the sequence of assembly only if the value of its operand is true. The operand consists of a Boolean expression enclosed in parentheses, followed by the sequence symbol of the statement to be processed next if the expression is true.

9. ANOP used to define a sequence symbol when one can not be placed on the desired statment. (i.e. the statement already contains a label or is a MEND card.)

10. ACTR redefines the maximum number of AGO and AIF

branches which can occur in a single macro call. Its operand is of the same format as the SETA statement. If an ACTR is specified, it must occur before the first AGO or AIF, and there may be but one ACTR per macro definition. If the count of AGO banches is exceeded, any subsequent AGOs are flagged and ignored.

The two macro pseudo-ops define the start and end of a macro definition. The prototype card defines the macro name and any associated dummy variables.

1. MACRO indicates the start of a macro definition.
2. MEND indicates the end of a macro definition.
3. Prototype Card a copy of the macro call card defining dummy arguments and the macro name. The macro name may be a maximum of five characters, the first of which must be alphabetic. If the name duplicates an assembler operation or pseudo-operation, no error will be generated but an attempt to call the macro will be treated as the assembler operation. A symbol may appear in the label field. If so, the label on the macro call card will replace that symbol in the macro definition when the macro is

called. Dummy arguments must be preceded by a dollar sign ($) and may appear in SET expressions with SET symbols.

For example:

```
        MACRO
$LABL   ADD     $REG,$LOC1,$LOC2
$LABL   L       $REG,$LOC1
        A       $REG,$LOC2
        MEND
```

Given the above macro definition the following line would call the macro and generate the lines which follow.

```
LOOP    ADD     3,ONE,TWO
LOOP    L       3,ONE
        A       3,TWO
```

Set Symbol Replacement

Set symbols and dummy parameters are replaced with their current values wherever they appear in source lines. Only lines containing conditional pseudo-ops and comment cards have no replacement performed upon them. Undefined symbols are not replaced. The set symbols are identified by the dollar sign which is their first character. They are terminated by any delimiter. Delimiters are:

$. + - * / ( ) , = and space

The period is used as a concatenation character. When it appears

as the terminating delimiter of a set symbol it will not appear in the expanded line. This feature makes it possible to concatenate a set symbol with a character string. A dollar sign ($) which is to appear in the output line must be represented as two consecutive dollar signs ($$). These will be interpreted as a single dollar sign and not as the first character of a set symbol.

The arithmetic set symbol is replaced by its numeric value. A negative number appears with its sign; a positive does not. Boolean set symbols are replaced by the symbol 1 if true and 0 if false. Dummy parameters are replaced by the character string which appears on the macro call card.
For example:

Suppose $A and $B are defined by the following two SET statements

```
$A      SETA   2
$B      SETA   5
```

Then the following list shows how various combinations of these symbols would be converted.

AB	becomes	25
$A.$B	becomes	25
$A.30	becomes	230
$$$A	becomes	$2
30$A	becomes	302
30.$A	becomes	30.2
$A..30	becomes	2.30

There are two system variables available to the user. The index ($SYSNDX) is a counter indicating the number of the current macro call. This number is unique to each macro call and constant throughout the call. If referenced at the source level its value is zero. In a macro call its value is a two digit integer which may be used to generate unique labels. The second system variable ($SYSECT) is the name of the control section at the time of the macro call. If the macro call changes the current section to generate data or instructions, this value may be used to return to the calling section before terminating the macro. To ensure uniqueness no user SET symbol or dummy parameter may begin with the characters $SYS.

Parameter substitution takes place in macro definitions defined in a macro call just as it would in any lines generated by a macro call. This allows variable macro definitions to be defined depending on the parameters specified in the calling statements of the macro call, however it also creates certain problems. In particular, variables which are local to the inner macro must not duplicate the names of variables in the outer macro. Also, global and system variables will be replaced when the inner macro is defined, not when it is called. To use these variables in such a macro they must be set to local variables and the local names used throughout the inner definition. This procedure works because SETA and SETB statements have no parameter substitution performed upon them.

Execution of LUIAS

LUIAS is designed to run as a batch program. It requires at least 40k of central memory. The format of the call card is:

LUIAS(input,output,pgm)

where:

input is the name of the input file. (If omitted INPUT is assumed.)

output is the name of the output file. (If omitted OUTPUT is assumed.)

pgm is the name of the file to which the object code is written. (If omitted PGM is assumed.)

A standard assembly from cards would require the following sequence:

```
(job card)
ATTACH(LUIAS,ID=XXX)
LUIAS.
CATALOG(PGM,ZZZZ,ID=YYY)
7-8-9

   assembly source cards

EOF
```

It is necessary to catalog PGM file if the program is to be be run on the simulator through INTERCOM.

3.2 Comparison of LUIAS to the IBM Assembler

This section outlines the major differences between the Lehigh University IBM System/360 Assembler and the IBM System 360 Assembler. Under each item is listed a brief description of how IBM handles the situation and how it is handled by LUIAS. Also listed for many of the items are the constraints placed upon the user by the limitations of LUIAS.

1) The LTORG statement

a) IBM - Literals referenced after the last LTORG and before the END card are stored at the end of the first CSECT.

LUIAS - Such literals are defined at the end of the current CSECT, instead of the first.

User Constraint - The user may insert a CSECT card for the first CSECT prior to the END card, so that the current CSECT when the END card is encountered is the first CSECT. If the last section is a DSECT, any remaining literals will be defined in the DSECT and thus will not occupy any storage.

b) IBM - Literals appear in an LTORG block only if

they are referenced in the same CSECT as the LTORG is in.

LUIAS - Literals appear in an LTORG block if they are referenced after the last LTORG. Inclusion is based solely on sequence, not CSECT.

User Constraint - The user must include an LTORG for every change of CSECT or allow the simultaneous addressability of every CSECT.

2) Set Symbols and Dummy Parameters

IBM - Set symbols are distinguished by an ampersand (&) as the first character.

LUIAS - These symbols are distinguished by a dollar sign ($) as the first character.

User Constraint - Two consecutive dollar signs ($$) must be used to represent a dollar sign in the source program to distinguish the dollar sign from the start of a set symbol.

3) System Index

IBM - &SYSNDX generates a four digit number if in a macro call; at the source level it is invalid.

LUIAS - $SYSNDX generates a two digit number; its value is 00 at the source level.

User Constraint - The two digit number allows unique five character labels to be generated.

4) &SYSLST

IBM - Dummy parameters may be referenced by position besides by name.

LUIAS - Not implemented.

5) Attributes

IBM - Various information about the definition of a label is available in macro SET expressions.

LUIAS - Not implemented.

6) ORG Statement

IBM - A blank operand returns the location counter to its previously highest value in the current section.

LUIAS - A blank operand is invalid.

User Constraint - The user must reset the location counter to its highest value before the end of the section. Failure to do so could result in some generated code being lost.

Example -

```
Location counter     Statement
        0120         A     DS    60C
        015C         B     DS    20C
        0170         HIGH  EQU   *
        0120               ORG   A
        0120               DC    C(BEGIN)
        0125               DC    55C( )
        015C               END
```

In this example, the last 20 bytes starting at B would be lost. These would be saved by inserting the following ORG statement before the END card.

```
ORG  HIGH
```

7) CSECT/DSECT Statements

IBM - A blank CSECT is valid; there is no limit to the number of CSECT or DSECT statements allowed in a program.

LUIAS - A blank CSECT is not allowed; there may be no more than eight CSECTs or eight DSECTs in a program.

8) DC/DS Statements

IBM - IBM allows a large variety of parameters to be specified. For full details, consult the IBM assembler manual (3).

LUIAS - The following is a list of restrictions in LUIAS assembler.

a) A constant in a DC statement may be a maximum of six characters long and may generate no more than five bytes.

b) Parentheses are used instead of apostrophes in all type constants.

c) Type constants E, S, V, and Y are not implemented. Type D is permitted only with a zero duplication factor.

d) Length sub-field is not implemented.

e) The duplication factor is not permitted when multiple operands are specified.

f) Character constants terminate with a comma or right parenthesis. The first character after the left parenthesis is always generated, even if it is a comma or right parenthesis.

g) Expressions is a DC A constant must evaluate as a valid relocatable or absolute address. If relative, the section of the address must be a CSECT, not a DSECT. In addition, no intermediate result may be invalid as the expression is processed from left to right.

h) Constants may not be contained in DS statements. The first character after the data type must be a space.

9) Literals

IBM - IBM permits any valid DC statement operand to be referenced as a literal.

LUIAS - Restrictions listed for DC/DS Statements under parts b, d, e, and f also pertain to literals.

a) No constant may consist of more than six characters between the parentheses.

b) A-type constants may only refernce single addresses; expressions or constants are prohibited.

c) Types D, E, S, V, and Y are not implemented.

10) Position of MACRO Definitions in the Source Deck

IBM - Macro definitions must occur before the START card.

LUIAS - Macro definitions may occur anywhere within the source deck.

User Constraints - The user must ensure that macro calls are defined before they are referenced. Macro definitions may occur within macro calls.

11) Continuation Cards

IBM - Any source card may be continued by placing a non-blank in column 72.

LUIAS - Only macro prototype statements and macro call statements may be continued.

User Constraints - Multiple source statements must be used to avoid continuation cards.

12) SETA Expressions

IBM - Any arithmetic expression which does not contain unary minus signs is valid.

LUIAS - Only a single arithmetic operand may appear in a SETA expression. Either term in the expression may be preceded by a unary minus. Expressions must appear within parentheses and may not include constants.

User Constraints - Complicated expressions may be expressed using multiple SETA statements.

13) SETB Expressions

IBM - A constant zero or one with or without parentheses, or any Boolean expression in parentheses. (LT, LE, EQ, NE, GE, GT, AND, OR, NOT are valid operators.)

LUIAS - A constant zero or one without parentheses or a Boolean expression containing a single operator only. NOT is permitted as a unary operator on any Boolean value appearing in an expression not containing relational operators. The exclusive or operator (XOR) is also permitted.

User Constraint - Complicated functions may be defined by using multiple SETB statements.

14) SETC Statement

IBM - Allows character constants to be defined. (There are also LCLC and GBLC statements.)

LUIAS - Not implemented

User constraints - Blank values are assigned initially to all GBL and LCL values and omitted parameters in macro call statements. Character constants may be assigned as parameters in macro calls. Only numeric values may be assigned to LCL and GBL variables by SET statements.

15) AIF

IBM - Any Boolean expression.

LUIAS - Any Boolean expression as restricted in SETB.

16) AGO/AIF

IBM - Assembly sequence may be transferred either forward or backward.

LUIAS - The sequence symbol must be located forward of the AGO or AIF at the source level; in a macro the symbol may be in either direction.

17) Statement Labels

IBM - Up to eight characters starting with an alphabetic character.

LUIAS - Only the first five characters are used; the remaining are ignored.

18) Expressions

IBM - Any expression may be formed with the arithmetic operators (+, -, *, /).

LUIAS - Only plus (+) and minus (-) are valid. The sequence of the terms in an expression is critical. The string, as processed from left to right, must never yield an invalid combination of terms. (See user's manual for details.)

User Constraint - The user must arrange terms of an expression in a valid left to right sequence.

3.3 Technical Notes on Modifications to the Assembler

This section describes the procedures taken in the adaptation of two independant programs into a unified macro-assembler. It describes the changes which had to be made to the two programs independently and then the changes which were required to interface the routines. This section is intended to provide an overview of the problem; it does not give detailed descriptions of individual routines.

The two programs mentioned above are the Lehigh University IBM 360 Assembler, written by Henry P. Seager (5), and a macro generator, written by George S. Maroudas (4).

During the 1974-1975 year two programs were written for use with the Lehigh University IBM 360 Simulator in the EE 315 class. The two, an assembler and a macro generator, made it possible to easily produce object code for use with the simulator.

Using the macro processor required a preassembly pass, however. An obvious improvement was to include the macro processing facility in the first pass of the assembler. This procedure has numerous advantages resulting in a more efficient and easier to use assembler.

Moreover, the assembler needed some enhancements; as originally implemented it included only a minimal subset of the IBM assembler pseudo-ops.

By combining the macro processor into the assembler execution time improves because less processing time is required. In a separate macro processor every statement must be searched against a macro name table as a possible macro call. This results in a large amount of wasted computing effort since the actual number of macro calls is usually small when compared to the number of statements in a program. When a macro processor is included in the first pass of an assembler, only those statements which are not recognized as assembler mnemonics need to be checked against the macro name table.

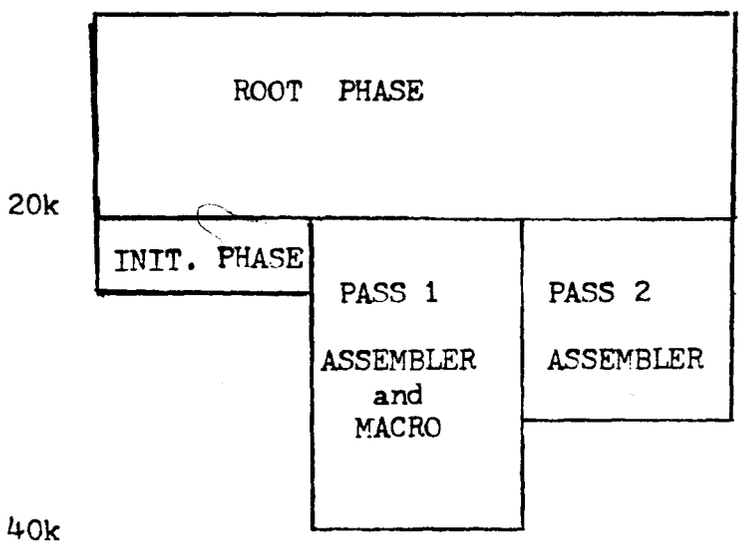
Also, pseudo operations associated with macro processing may be recognized by the assembler pseudo-op handler rather than

by a special processor.

A drawback to including the macro processor in the assembler is the overall size of the resultant program. Had the two existing programs been combined as originally written the resulting program would have required a load size of nearly 100k words of central memory. Such a memory requirement would have greatly reduced the assembler's availability under the contraints currently imposed by the Lehigh University Computing Center. (In particular, large jobs receive low priority and thus slow turn-around time.) To reduce the load size the assembler was modified to use overlays. As much as possible, code which was only used during pass two was removed from the first pass. The size of the second pass was not as critical because most of the macro processing was to be added to the first pass.

The resulting program consists of four segments: a root phase and three overlays. The root phase handles dayfile messages and loads the three overlays as they are needed. In addition, it contains a number of routines, including the I/O routines, which are used by the three overlays. The first overlay contains only some initialization code and hence is quite small. The second overlay is pass one of the assembler. This is the largest of the segments, containing the macro processing code in addition to the pass one assembler code. The third overlay is pass two of the assembler. This pass

generates the code and all the listings produced by the assembler. The overlay structure of the assembler is portrayed in the following table.



The biggest saving in memory size in the first pass came from the removal of the array ICONT from the first pass. This array is used for storing the object code produced by the assembler during the second pass. During the first pass it was referenced in some places only because the code was identical for both passes. These references were removed so that the array was not needed at all in the first pass, thus saving 2500_{10} words.

Additional savings were obtained by changing the manner in which decisions were made regarding different actions for the two passes. Originally the assembler tested at execution time and branched to appropriate routines depending on the pass. The addition of comditional assembly statements to the LUIAS source deck allows the COMPASS assembler to generate, from common source,

two different routines for the two passes.

A number of problems in the assembler were corrected, in addition to numerous enhancements made, as the assembler was converted to an overlay structure.

There were two problems in handling expressions in instruction operands. Location counter references were not always recognized and the relocatibility of an expression was calculated incorrectly. If any term appearing in an expression was relative, the entire expression was considered relative. In correcting this, a limitation was placed upon assembler expressions. Before each operation is performed, as the expression is processed from left to right, the terms are checked for relocatibility; if both are relocatible the statement is flagged as an error.

There were also two problems in DS and DC statements. If a valid expression was found in a DC or DS statement, processing stopped immediately without checking any more characters on the card. Thus a user who tried to enter a complicated, but invalid expression might not receive an error message indicating that the assembler did not fully process the statement. Similarily, in a C or Z type constant, attempting to generate a six byte constant would produce an invalid constant but not an error message.

The ORG statement previously required an absolute expression in its operand rather than a relocatible expression. Correcting this was necessary for ORG statements to function properly in multi-CSECT programs.

A number of changes were made to the assembler to make it more powerful. These included the addition of new pseudo-ops and modifications to the assembler listings produced.

The CNOP instruction was added. This instruction advances the location counter from zero to six bytes until the location counter is on a specified boundary. The bytes skipped are filled NOPR instructions.

CSECT and DSECT pseudo-instructions were also added. These permit code to be generated in independant control sections and for data area formats to be defined without reserving storage.

The DS statement was modified to specify boundary alignment without reserving storage. To this end a zero duplication factor is now permitted. Also, a D-type constant is permitted to align with a doubleword boundary.

An A-type constant is permitted in DC or DS statements. Previously, this type was only permitted in literals. This constant is evaluated by the same routine which evaluates expressions and thus is subject to the same restrictions. Also it is not possible to generate an address constant for a location within a DSECT because no storage is actually reserved for DSECTs.

Previously the ORG statement would only move the location counter forward. This routine was modified to allow the location counter to be moved in either direction. This necessitated resetting the pointers associated with the object code output

routines. This facility was also needed for CSECT processing, thus a subroutine was written which reset the output pointers to agree with the location counter value.

The character conversion routine for converting C-type data constants was changed to handle more of the CDC character set and to agree with the character conversion performed by the simulator. Previously only the IBM 026 keypunch character set was accepted; now the entire SCOPE 63-character set is accepted.

The format of the SIO command was changed to accommodate the more flexible I/O now performed by LUIS. Formerly the SIO command had a specially defined format which would handle only two I/O functions. It now has the standard SI format and any of 256 I/O codes may be specified, although only four are used by the simulator.

Two changes affect the object code written by the assembler. In addition to the code, the starting address and length are also written to the PGM file. The simulator makes use of this feature to eliminate some user type-ins. The object code listing was changed to improve its readability. It now lists the bytes broken up into words and includes the starting address of each line of code printed.

Two dayfile messages were added to allow the user to tell the results of the assembly at a glance. These messages give the number of assembler errors and the total time, in CPU seconds, required for the assembly.

By far the largest change to the assembler was the addition of the macro processor. Inasmuch as the macro processor was designed as a general purpose processor, a number of changes were made to provide more IBM-like processing.

A number of minor changes involving input format were made. The macro processor recognized continuation by a plus sign as the last non-blank character on a card. This was changed to a non-blank in column 72 of the input card. The processor was also converted to leave columns 73 to 80 unchanged, so they could be used for sequence numbers. The macro processor had formats slightly different from the IBM formats for SET, AGO, and AIF statements. These were changed for compatibility with the IBM formats. The macro processor used an apostrophe to distinguish dummy parameters. In the IBM assembler, the apostrophe is used for other purposes and an ampersand is used for dummy parameters. Since there is no ampersand (&) in the CDC SCOPE character set, a dollar sign ($) was used instead. Also, a minor bug, dealing with the handling of dummy parameters during recursive macro calls, was fixed.

A more complicated procedure was the interface of the macro processor routines into the assembler. In the macro processor a single routine read and wrote input cards, processed macro definitions, and recognized macro calls. In LUIAS, the macro pseudo-ops are recognized by the main assembler routines and appropriate subroutines called to process them. When an

undefined opcode is detected, a separate routine is called to look up this opcode in the macro name table. This routine generates the necessary instructions from the macro definition table and returns control when the expansion of the macro call is complete.

The macro processor required an inordinately large array for storing macro definition cards. This 8000_{10} (20000_8) word array stored the exact card images of up to 1000 cards appearing in macro definitions. This array was replaced with a mass-storage, random-access file. Taking into account the I/O routines and buffers required, the net savings were approximately 12k words.

Error handling also had to be modified slightly. Macro errors were handled by a common subroutine which printed an error message and then either returned control if the error was non-fatal or terminated the program if the error was fatal. Because the assembler should not be aborted during the first pass unless a table overflows, and because the program listing is not produced until pass two, the error routine was modified so that it only sets an error number and increments an error count. The error number and other macro flags are written to the temporary file along with the source card image. During the second pass, the error number is printed with the source line.

A number of values had to be added to the macro call

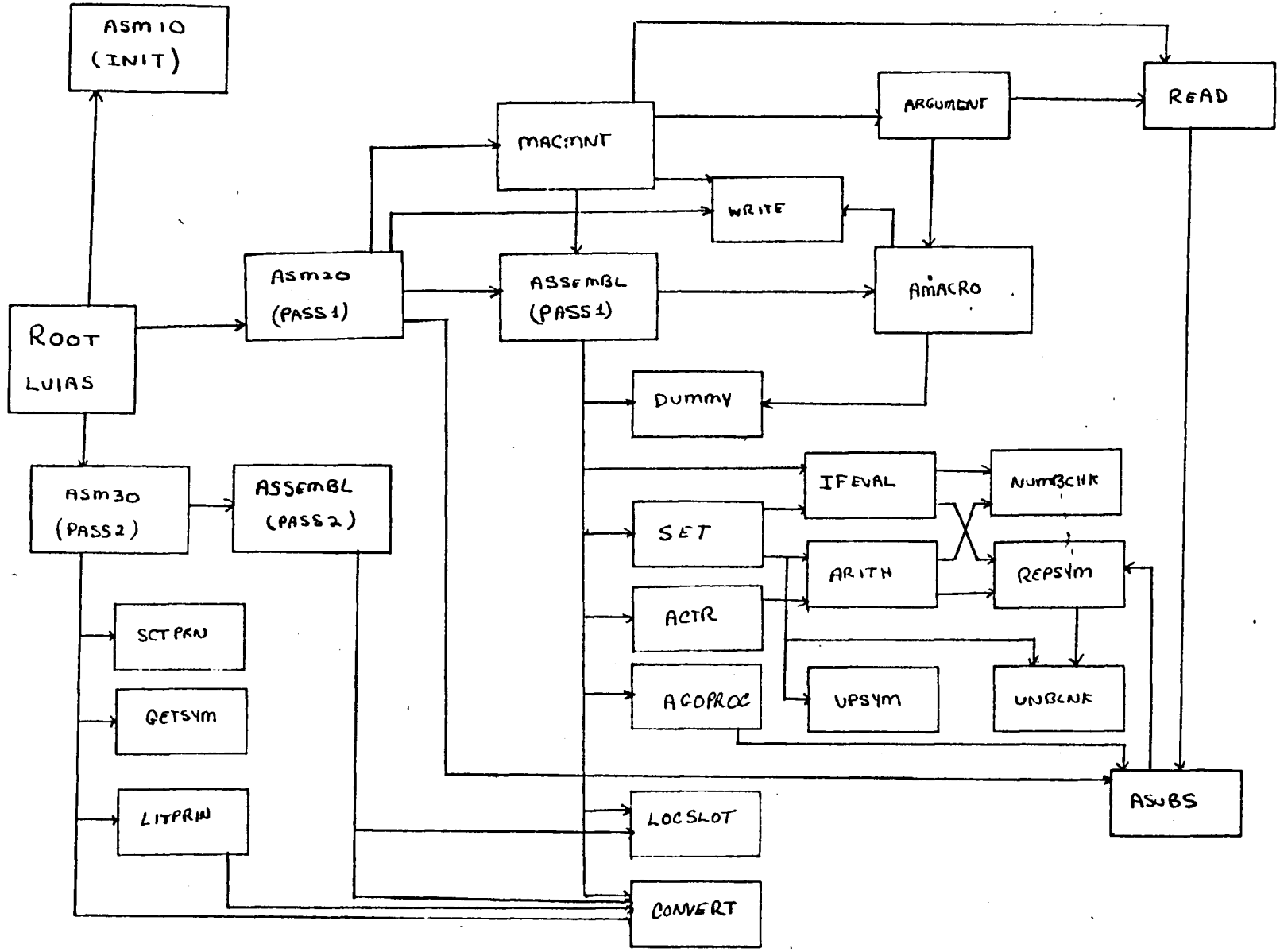
stack because more complicated procedures required more call dependant parameters. In particular, the ACTR value, the $SYSECT value at the time of the call, the $SYSNDX value for the call, and the index value in the macro name table must be stored in the stack.

In addition to the dummy parameters on the macro prototype statement, a label and local and global values may also be defined. A symbol appearing in the label field of the macro prototype statement is treated as a dummy parameter. It is replaced by the symbol on the macro call statement. Local and global values may also be defined. These are treated like dummy parameters except that they can only be defined through SETA and SETB statements.

3.4 Descriptions of the Routines and Common Blocks of LUIAS

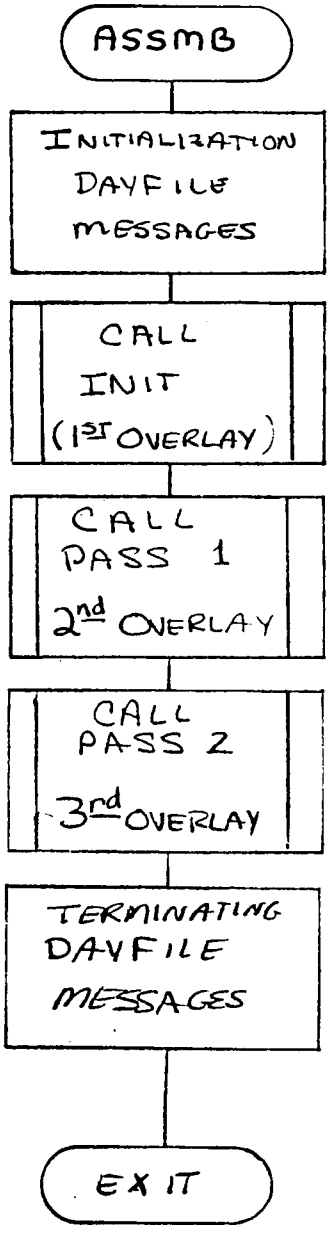
This section will give detailed descriptions of the routines added and changes made to LUIAS in the implementation of macro processing and the other enhancements described in this thesis. It will also describe the contents of the additions to the data bases used by the assembler.

Figure 3-1 is a flowchart showing the interaction of the various routines of LUIAS. Arrows connecting boxes point from the calling routine to the called subroutine. For clarity, several routines (AERROR, ABORT, WORDS) which perform general functions have been omitted.


ASM10
(INIT)
ROOT
LUIAS
ASM20
(PASS1)
ASM30
(PASS2)
ASSEMBL
(PASS1)
ASSEMBL
(PASS2)
MACINT
ARGUMENT
READ
WRITE
AMACRO
DUMMY
IFEVAL
NUMBCHR
SET
ARITH
REPSYM
ACTR
AGOPROC
UPSYM
UNBLNK
ASUBS
SCTPRN
GETSYM
LITPRN
LOCSLOT
CONVERT

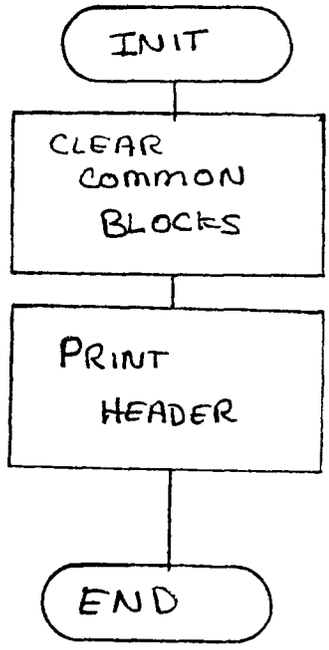
3.4.1 Program ASSMB

This Fortran routine is the main program of LUIAS. Its function is to call the three overlays in sequence. Common blocks are included in this program to ensure that they reside in memory at all times. This routine also enters messages in the dayfile at the start and end of the program. Files are defined in this routine and hence I/O buffers reside in the root phase.


ASSMB
INITIALIZATION
DAYFILE
MESSAGES
CALL
INIT
(1ST OVERLAY)
CALL
PASS 1
2nd OVERLAY
CALL
PASS 2
3rd OVERLAY
TERMINATING
DAYFILE
MESSAGES
EXIT

3.4.2 Program INIT

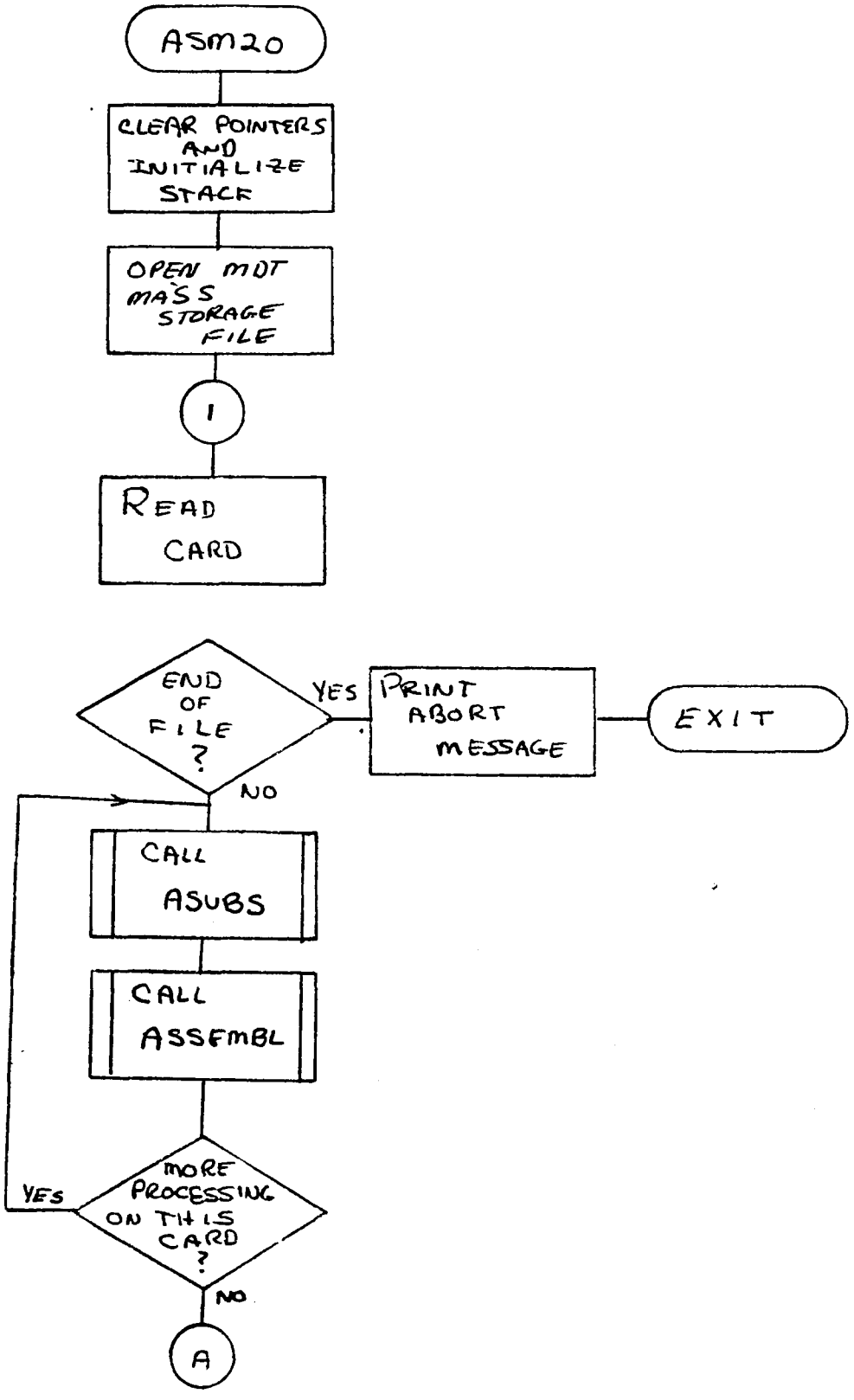
This Fortran routine performs minor initialization of some common blocks. Although this program is quite small it helps to reduce the critical size of the first pass.

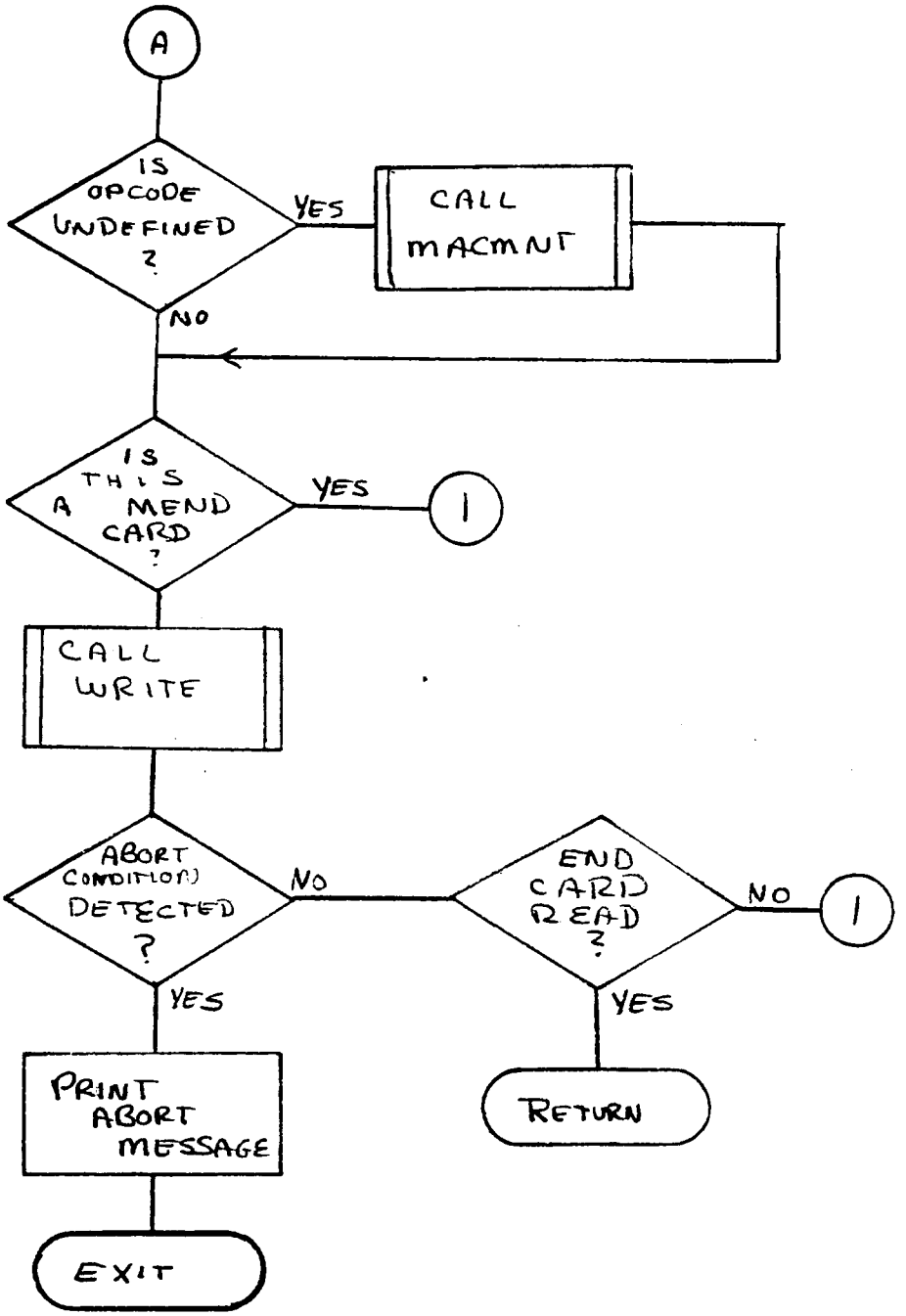


3.4.3 Program ASM20

This Fortran routine is the control portion of pass one. It reads cards from the INPUT file, calls ASSEMBL to assemble each card, and then writes the card to the temporary file. If a potential macro call is found by ASSEMBL a flag is set for ASM20, which calls MACMNT to process the macro call.

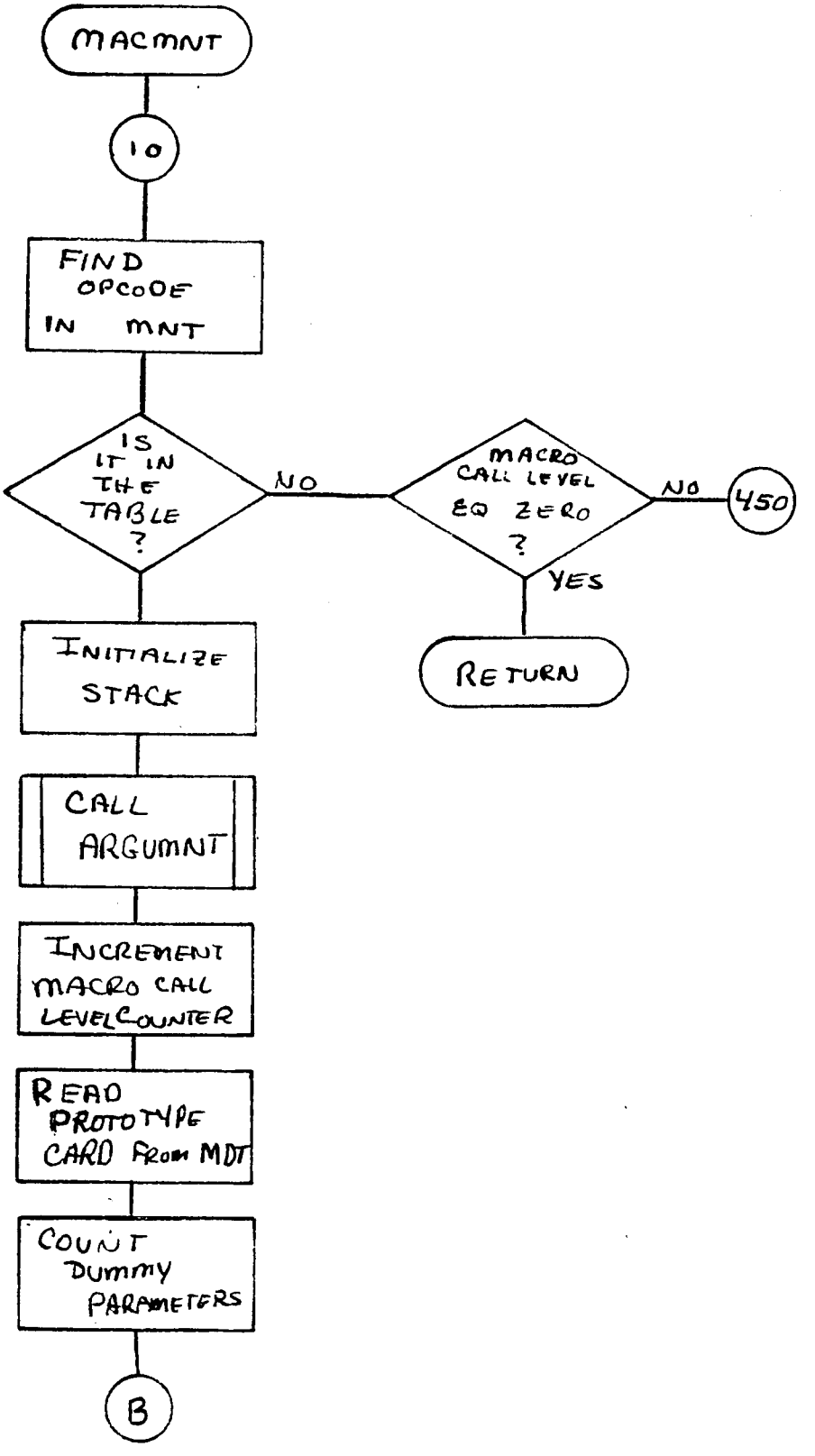
The NOSAVE indicator is tested to tell if the current card must still be processed or if it should not be saved for the second pass. If an abort condition is detected by ASSEMBL or if an EOF is detected before an END card, the program is aborted. When an END card is detected control is returned to the root phase.

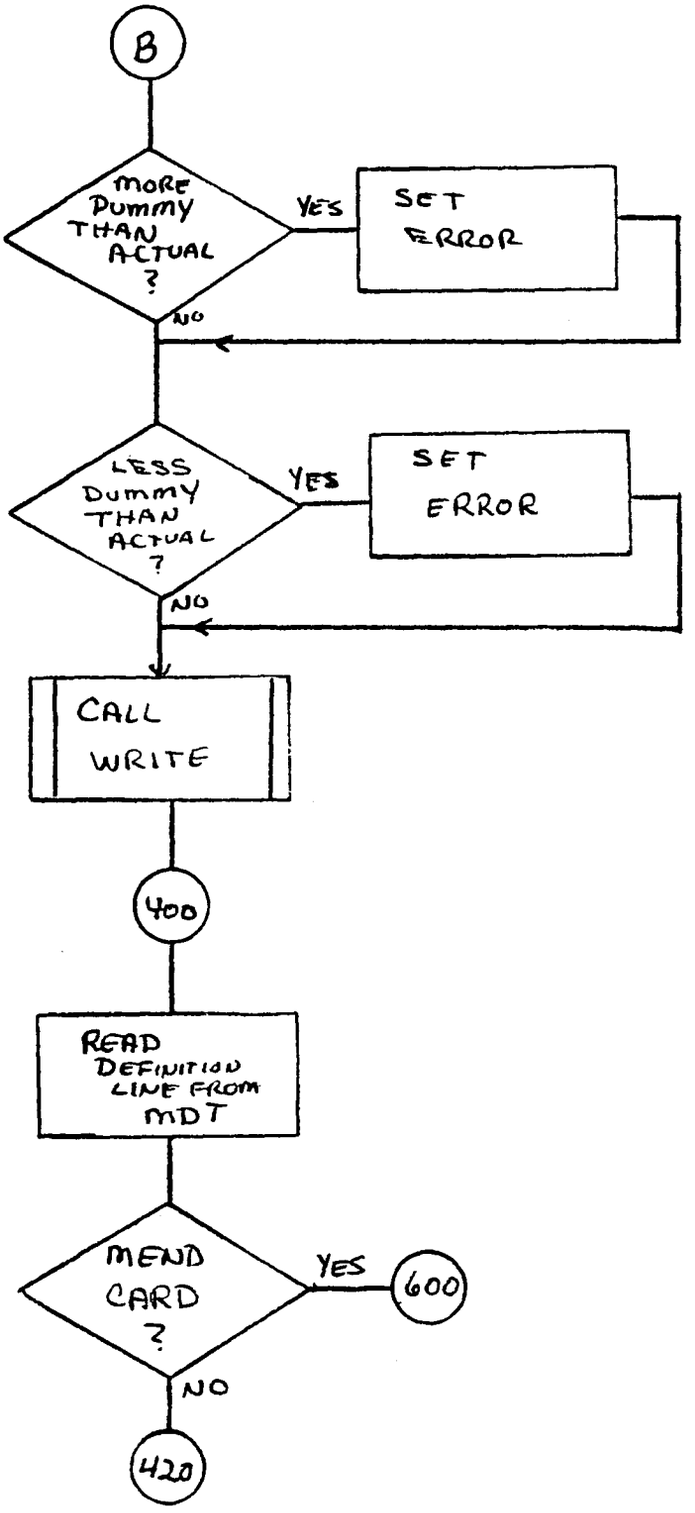

ASM20
CLEAR POINTERS AND INITIALIZE STACK
OPEN MDT MASS STORAGE FILE
1
READ CARD
END OF FILE ?
YES
PRINT ABORT MESSAGE
EXIT
NO
CALL ASUBS
CALL ASSEMBL
MORE PROCESSING ON THIS CARD ?
YES
NO
A

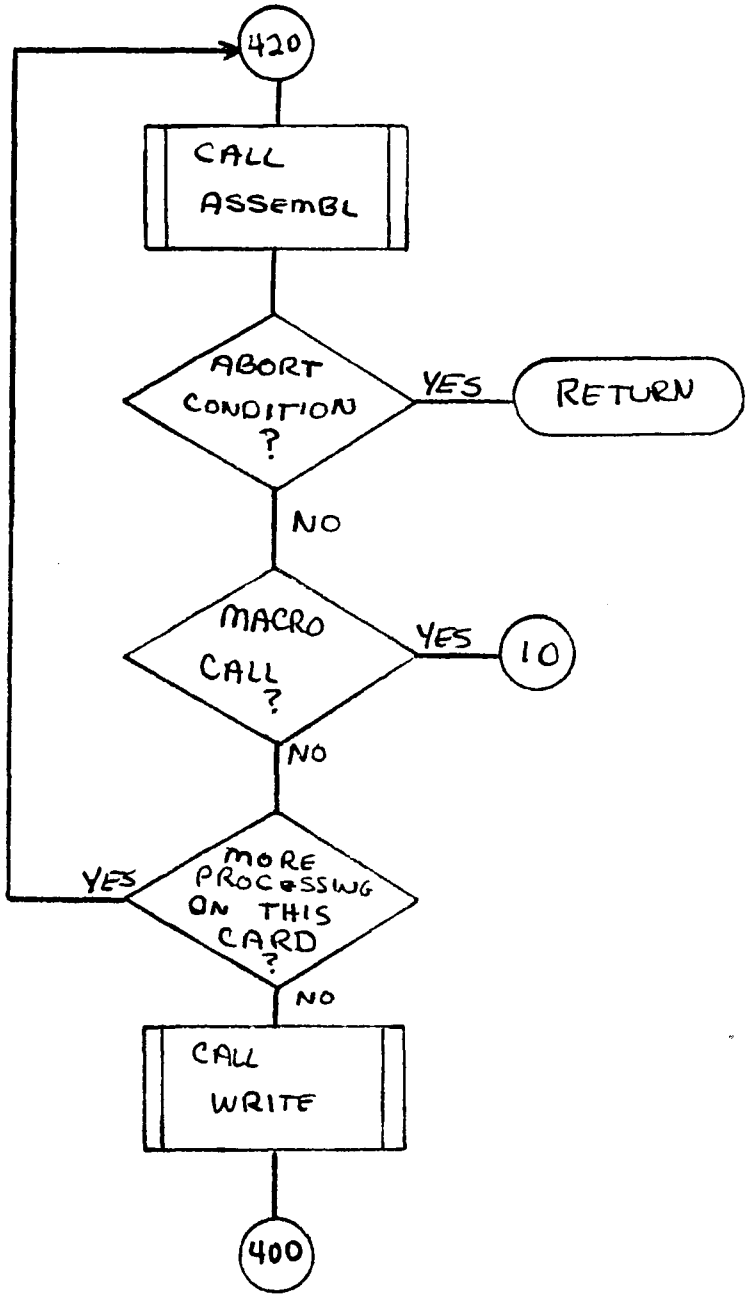

A
IS OPCODE UNDEFINED ?
YES
CALL MACMNT
NO
IS THIS A MEND CARD ?
YES
1
CALL WRITE
ABORT CONDITION DETECTED ?
NO
END CARD READ ?
NO
1
YES
YES
PRINT ABORT MESSAGE
RETURN
EXIT

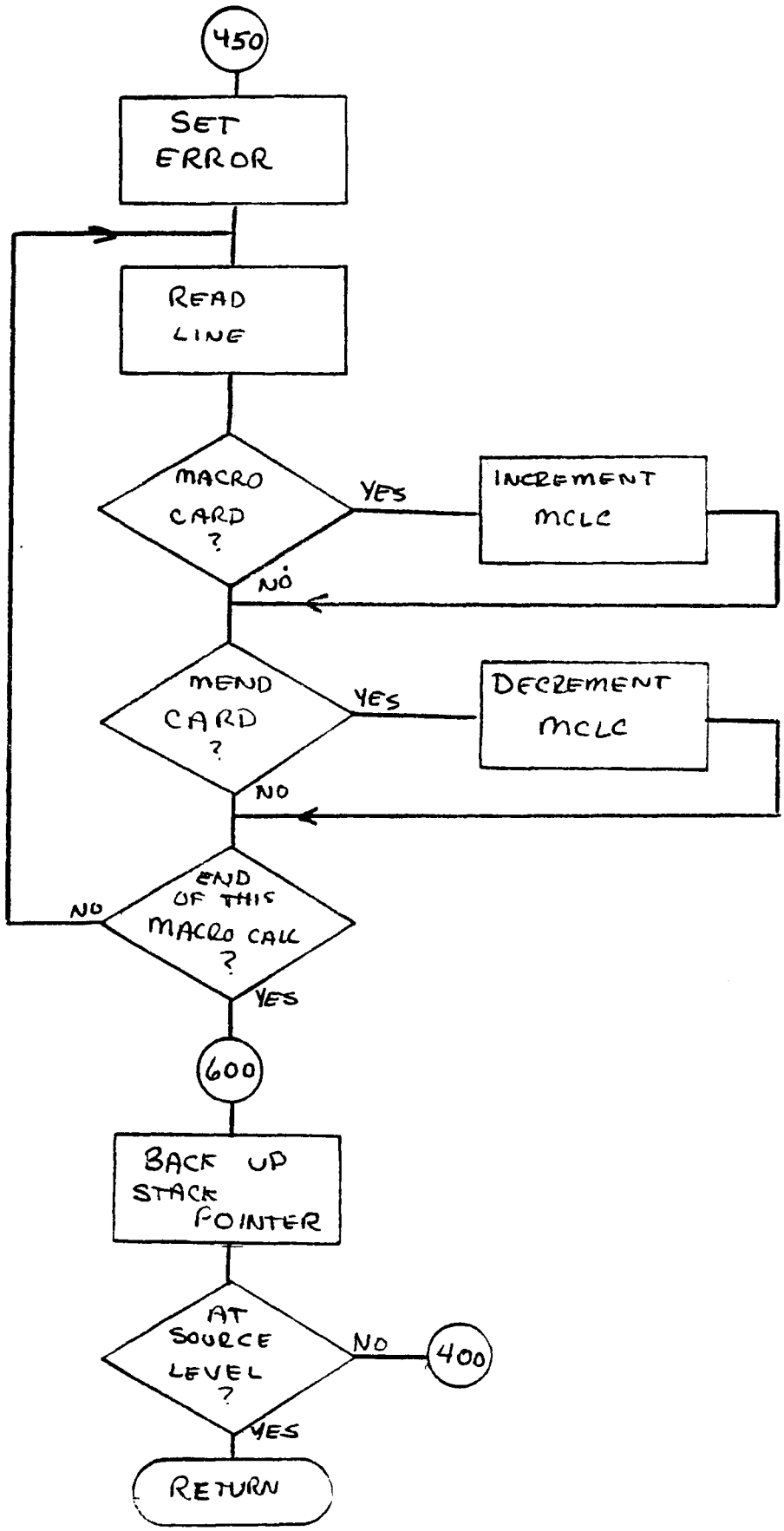
3.4.4 Subroutine MACMNT

This Fortran routine receives control when an undefined opcode is detected in ASSEMBL. If the opcode is not a macro call, control is returned and the line is treated as an error. If the opcode is a macro call, subroutine ARGUMNT is called to enter the actual arguments into the stack. Then lines are read from the Macro Definition Table (MDT) and passed to ASSEMBL for processing. If a MEND card is found control returns to ASM20. If ASSEMBL detects an undefined opcode, it sets a flag and MACMNT branches back to its entry point to effectively form a recursive call. Macro definitions within macro calls are processed without special effort, because these are handled by ASSEMBL independent of MACMNT.


MACMNT
10
FIND OPCODE IN MNT
IS IT IN THE TABLE ?
NO
MACRO CALL LEVEL EQ ZERO ?
NO
450
YES
RETURN
INITIALIZE STACK
CALL ARGUMNT
INCREMENT MACRO CALL LEVELCOUNTER
READ PROTOTYPE CARD FROM MDT
COUNT DUMMY PARAMETERS
B

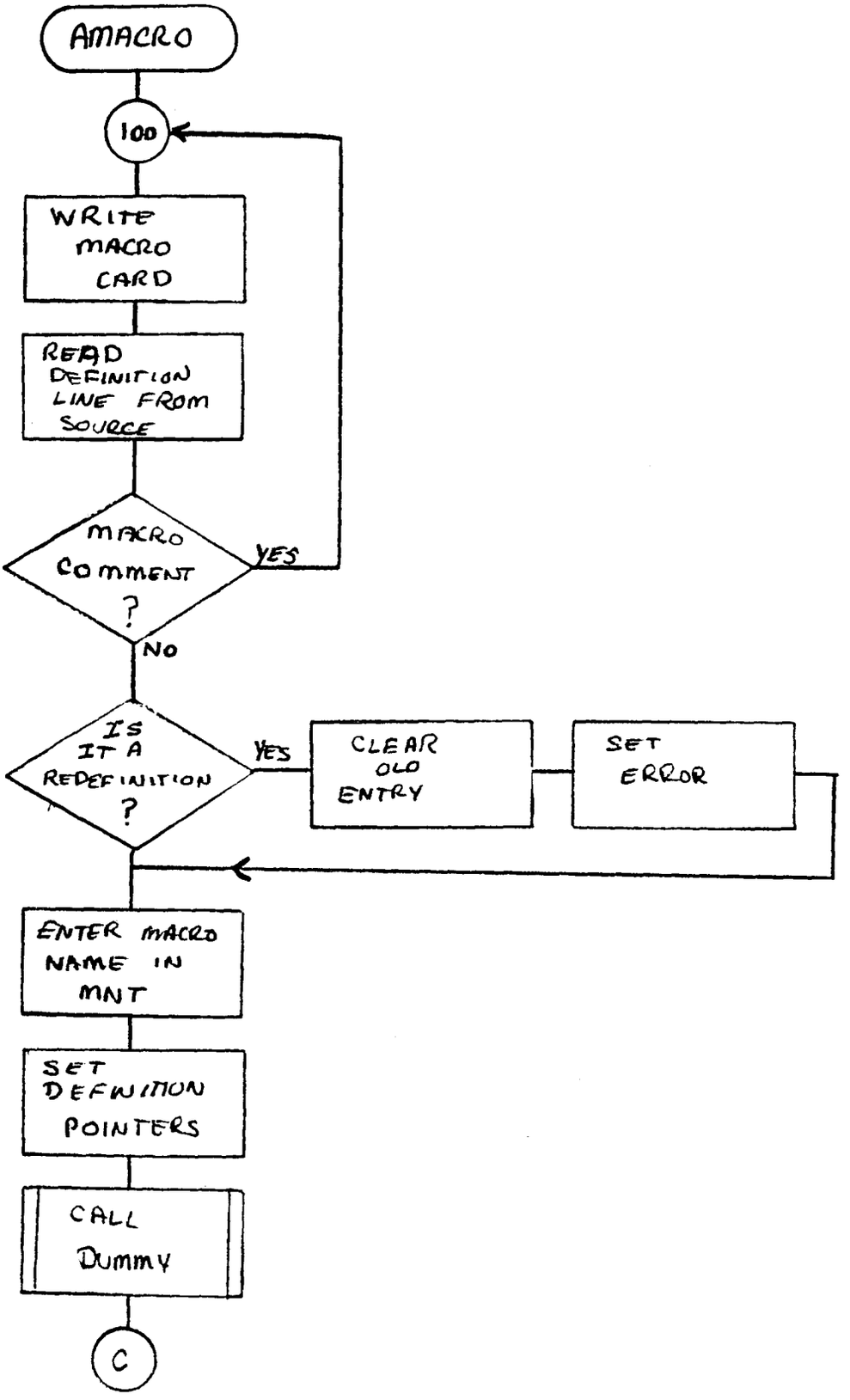

B
MORE DUMMY THAN ACTUAL ?
YES
SET ERROR
NO
LESS DUMMY THAN ACTUAL ?
YES
SET ERROR
NO
CALL WRITE
400
READ DEFINITION LINE FROM MDT
MEND CARD ?
YES
600
NO
420

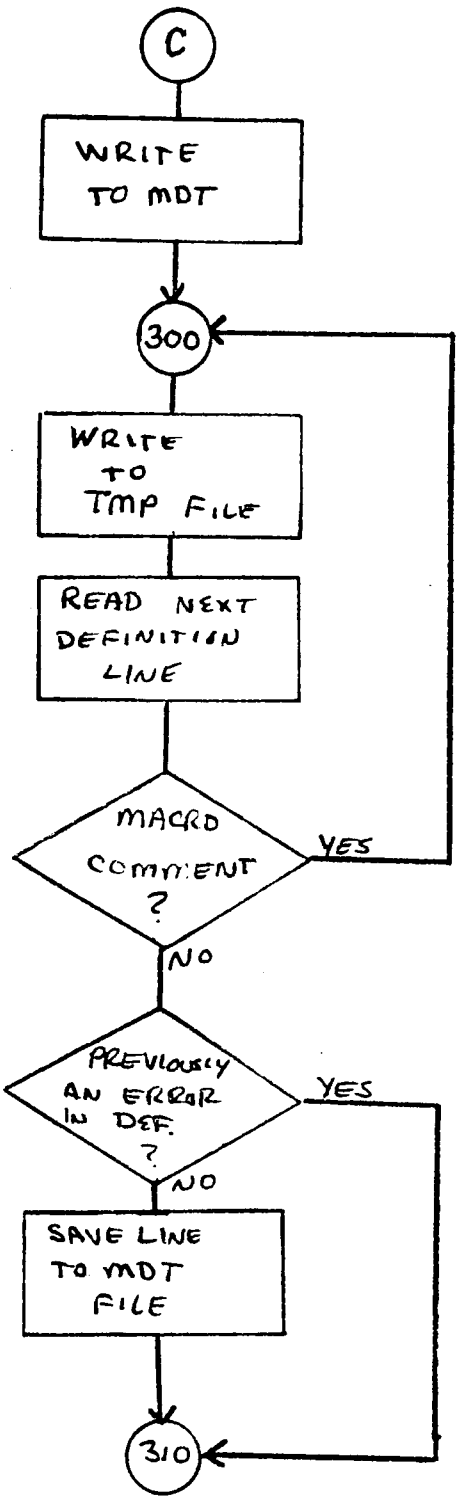

420
CALL
ASSEMBL
ABORT
CONDITION
?
YES
RETURN
NO
MACRO
CALL
?
YES
10
NO
MORE
PROCESSING
ON THIS
CARD
?
YES
NO
CALL
WRITE
400

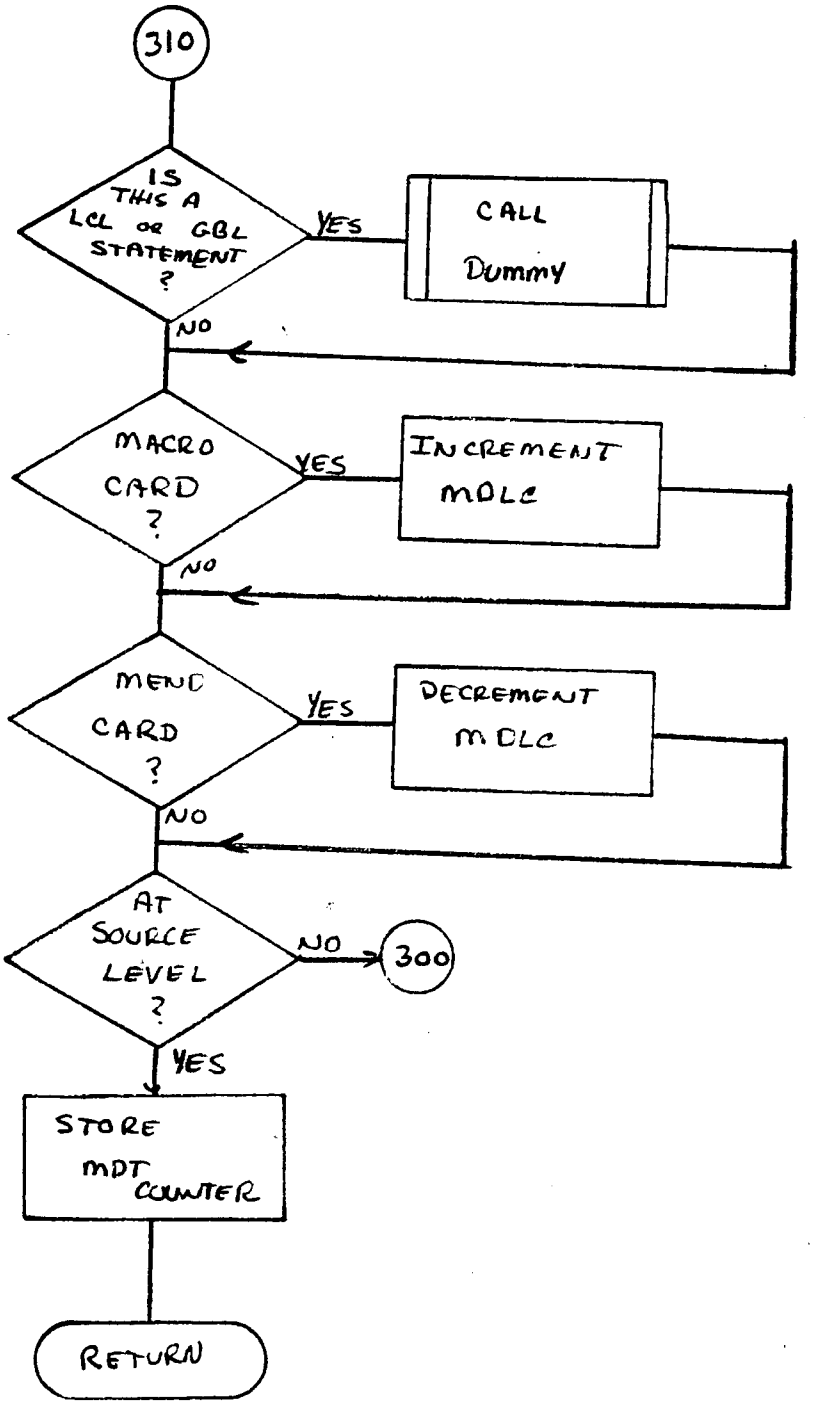

450
SET ERROR
READ LINE
MACRO CARD ?
YES
INCREMENT MCLC
NO
MEND CARD ?
YES
DECREMENT MCLC
NO
END OF THIS MACRO CALL ?
NO
YES
600
BACK UP STACK POINTER
AT SOURCE LEVEL ?
NO
400
YES
RETURN

3.4.5 Subroutine MACRO

This Fortran subroutine processes macro definitions. When ASSEMBL detects a MACRO card it calls MACRO. MACRO stores all cards between the MACRO card and its associated MEND card in the Macro Definition Table file (MDT). It calls DUMMY to store the dummy parameters on the macro prototype card in the Argument List Array (ALA1). It also calls DUMMY to process LCL and GBL statements. It allows for nested macro definitions by keeping a counter of MACRO cards which is decremented for every MEND card.

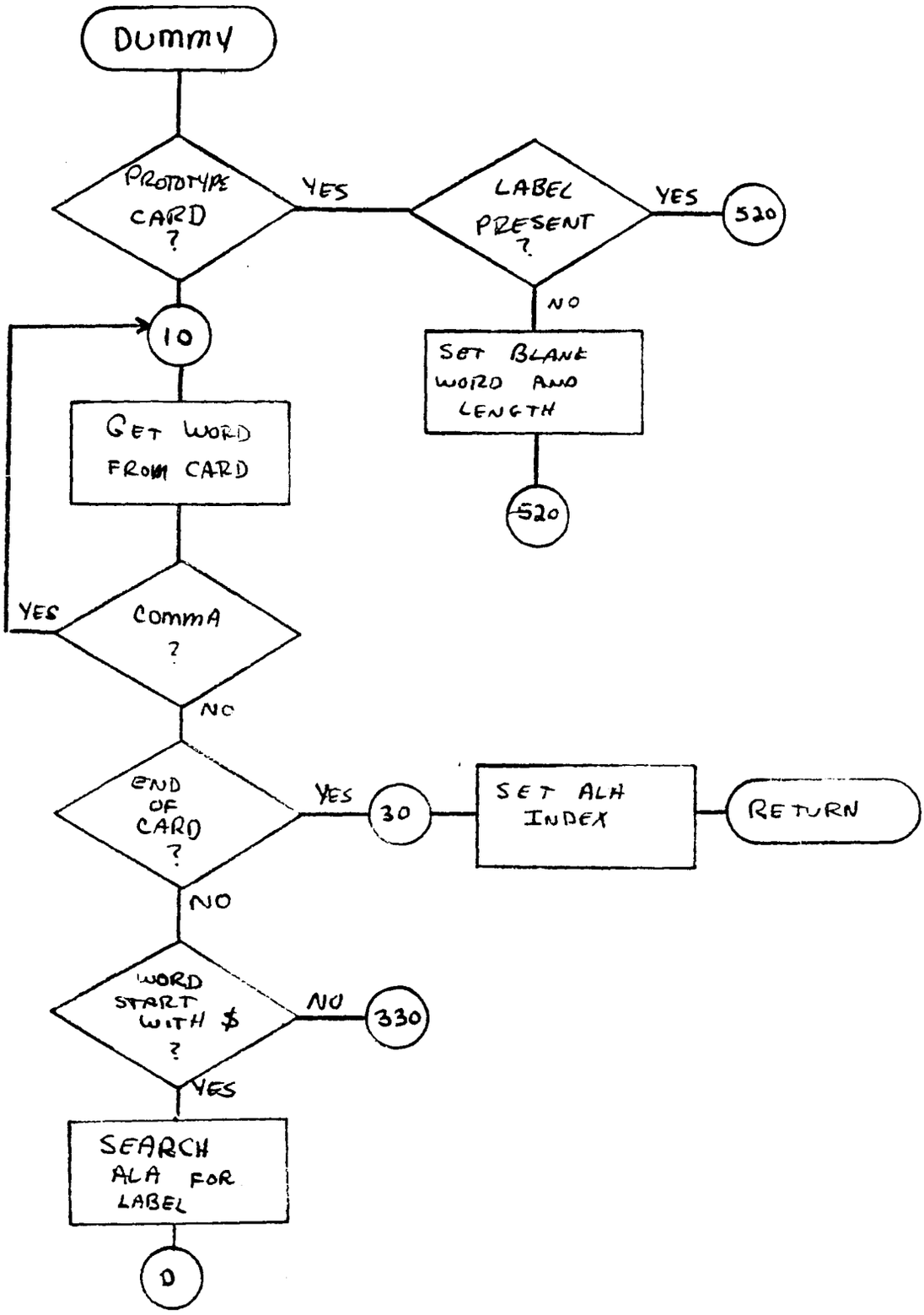

AMACRO
100
WRITE MACRO CARD
READ DEFINITION LINE FROM SOURCE
MACRO COMMENT ?
YES
No
IS IT A REDEFINITION ?
YES
CLEAR OLD ENTRY
SET ERROR
ENTER MACRO NAME IN MNT
SET DEFNITION POINTERS
CALL Dummy
C

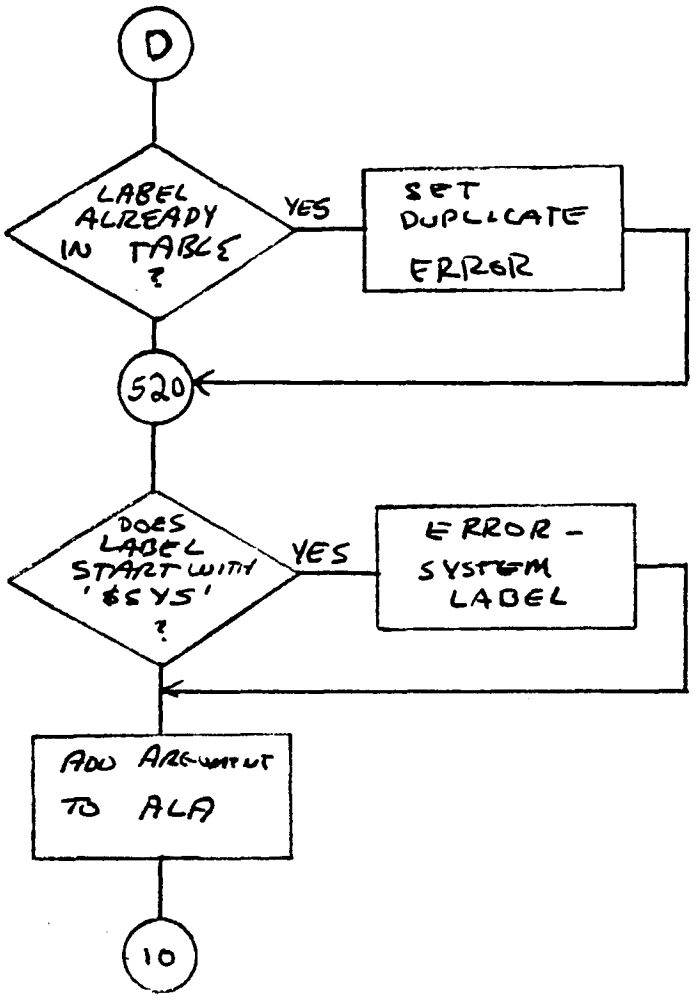

C
WRITE TO MDT
300
WRITE TO TMP FILE
READ NEXT DEFINITION LINE
MACRO COMMENT ?
YES
NO
PREVIOUSLY AN ERROR IN DEF. ?
YES
NO
SAVE LINE TO MDT FILE
310

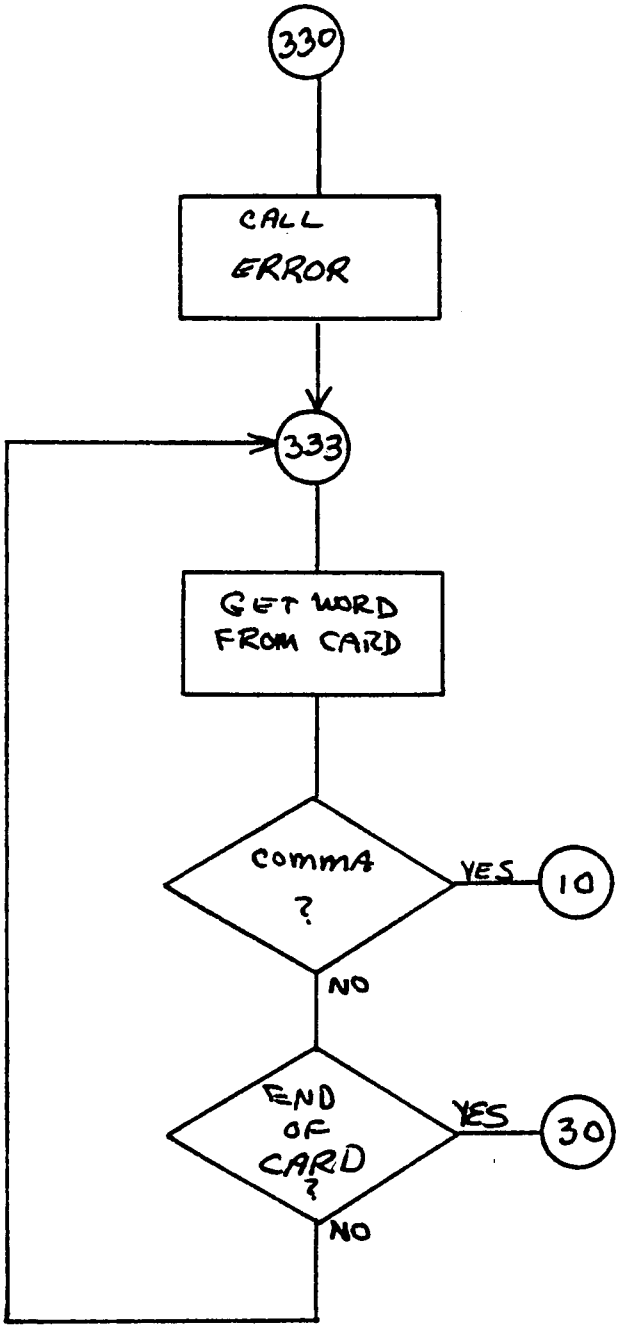

310
IS THIS A LCL or GBL STATEMENT ?
YES
CALL Dummy
NO
MACRO CARD ?
YES
INCREMENT MDLC
NO
MEND CARD ?
YES
DECREMENT MDLC
NO
AT SOURCE LEVEL ?
NO
300
YES
STORE MDT COUNTER
RETURN

3.5.6 Subroutine DUMMY

This Fortran routine enters dummy arguments from the macro prototype card and from LCL and GBL statements into the Argument List Array. It accounts for the possibility of a label on the prototype card and handles continuation cards.

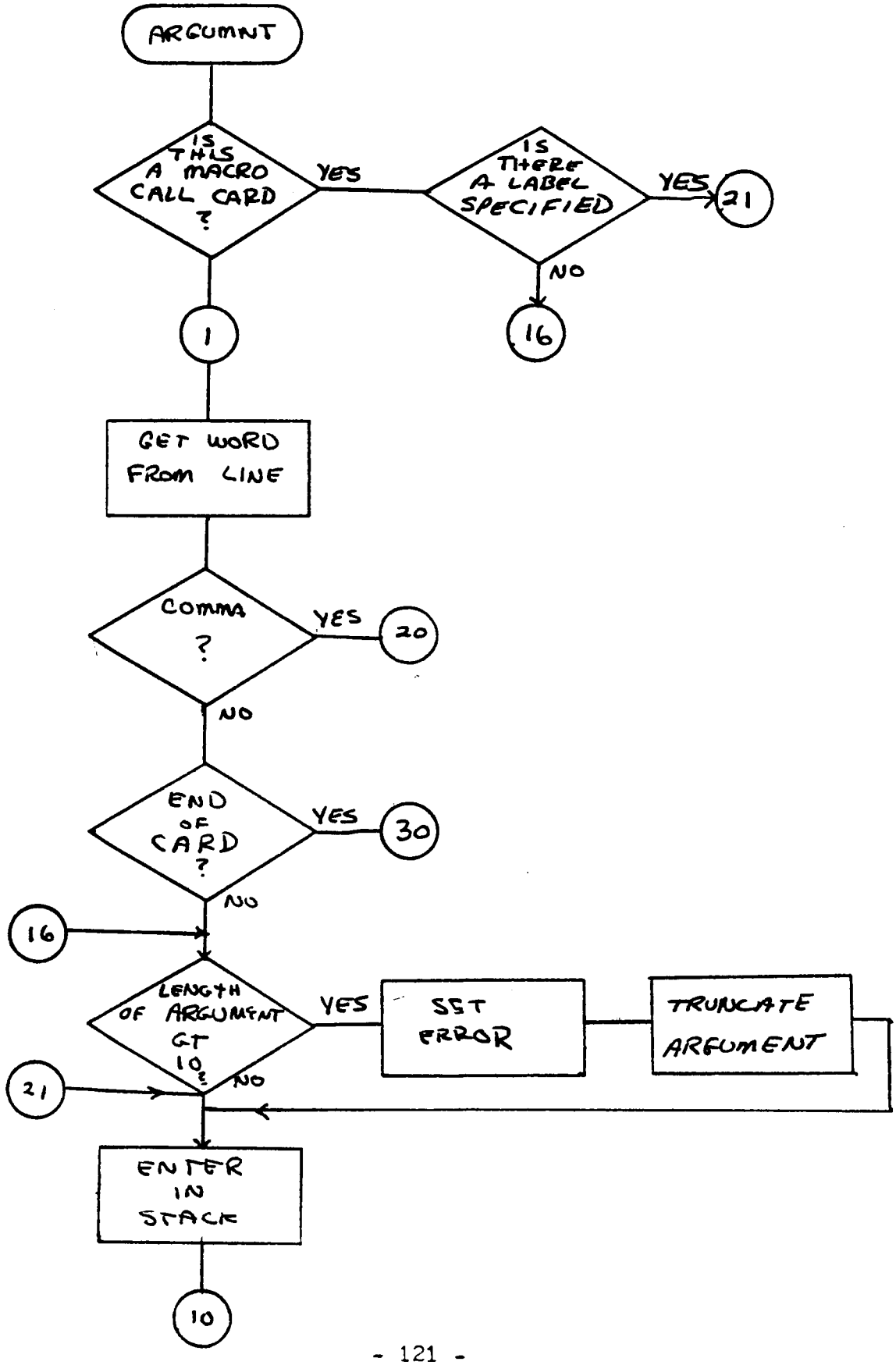

DUMMY
PROTOTYPE CARD ?
YES
LABEL PRESENT ?
YES
520
NO
SET BLANK WORD AND LENGTH
520
10
GET WORD FROM CARD
YES
COMMA ?
NO
END OF CARD ?
YES
30
SET ALH INDEX
RETURN
NO
WORD START WITH $?
NO
330
YES
SEARCH ALA FOR LABEL
D

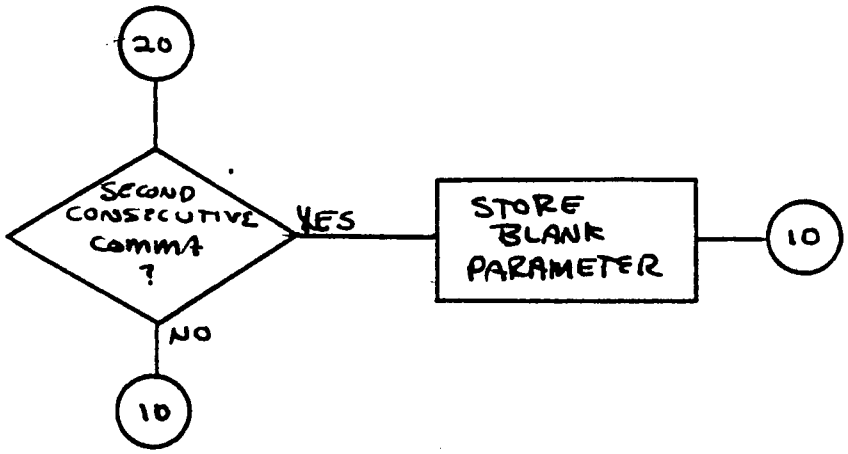

D
LABEL ALREADY IN TABLE ?
YES
SET DUPLICATE ERROR
520
DOES LABEL START WITH '$SYS' ?
YES
ERROR - SYSTEM LABEL
ADD ARGUMENT TO ALA
10

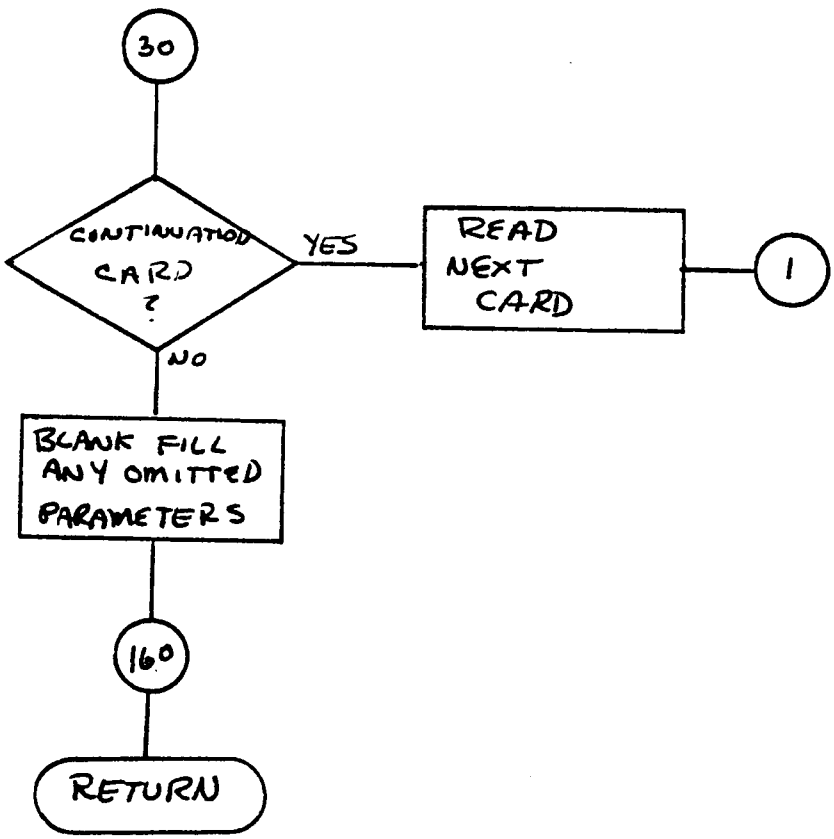

330
CALL
ERROR
333
GET WORD
FROM CARD
COMMA
?
YES
10
NO
END
OF
CARD
?
YES
30
NO

3.4.7 Subroutine ARGUMNT

This Fortran routine enters the actual parameters from the macro call line into the stack. It accounts for the possibility of a label as a dummy parameter. It also allows for the continuation of the macro call line.

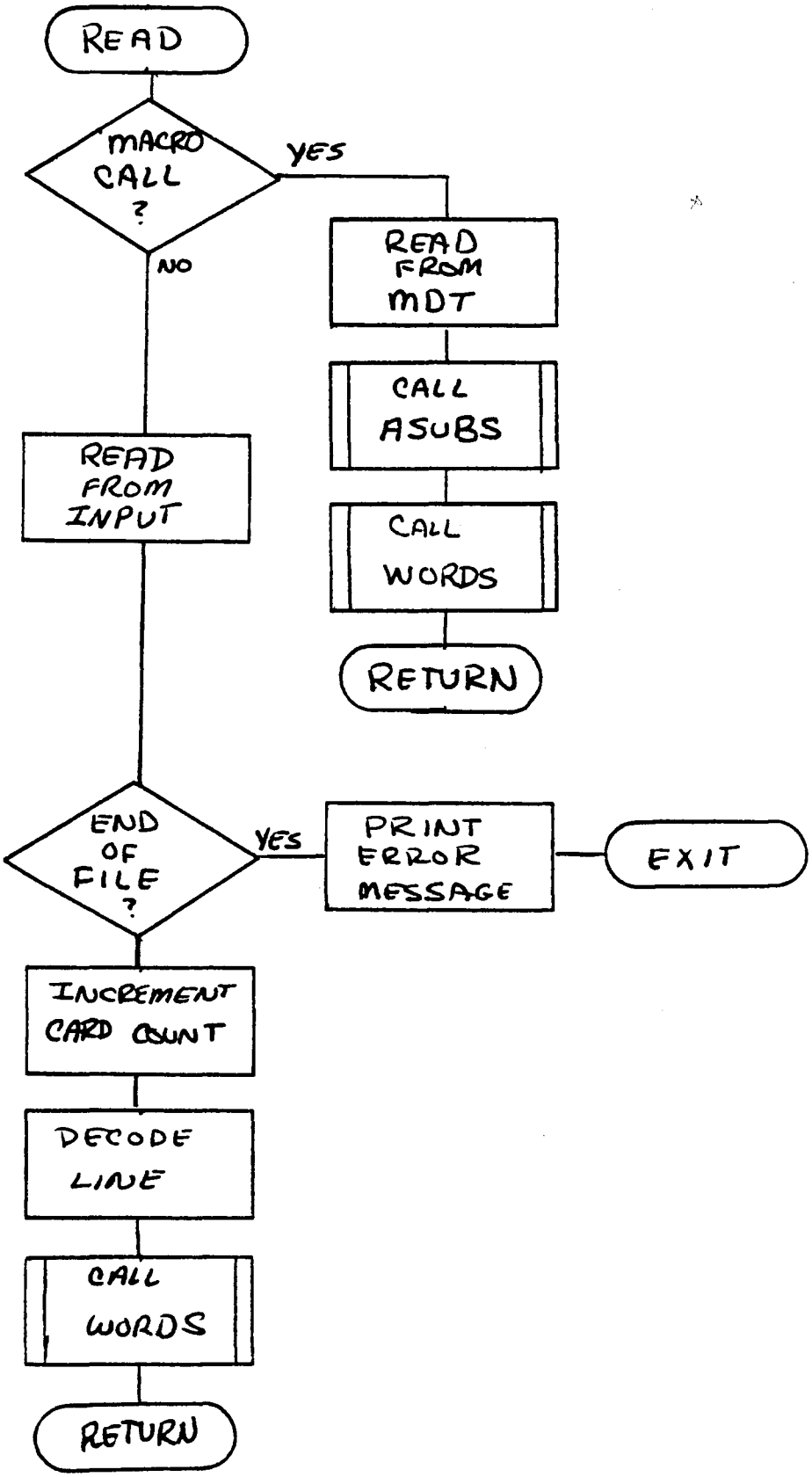

ARGUMNT
IS THIS A MACRO CALL CARD ?
YES
IS THERE A LABEL SPECIFIED
YES
21
NO
16
1
GET WORD FROM LINE
COMMA ?
YES
20
NO
END OF CARD ?
YES
30
NO
16
LENGTH OF ARGUMENT GT 10 ?
YES
SET ERROR
TRUNCATE ARGUMENT
21
NO
ENTER IN STACK
10


20
SECOND CONSECUTIVE COMMA ?
YES
STORE BLANK PARAMETER
10
NO
10


30
CONTINUATION CARD ?
YES
READ NEXT CARD
1
NO
BLANK FILL ANY OMITTED PARAMETERS
160
RETURN

3.4.8 Subroutine READ

This Fortran routine reads a source line from either the MDT file or from the INPUT file. It reads from MDT whenever a macro call is being expanded; at all other times it reads from input. It calls ASUBS to replace SET symbols when a macro call is being expanded. It also calls WORDS to get the first word from each line read.

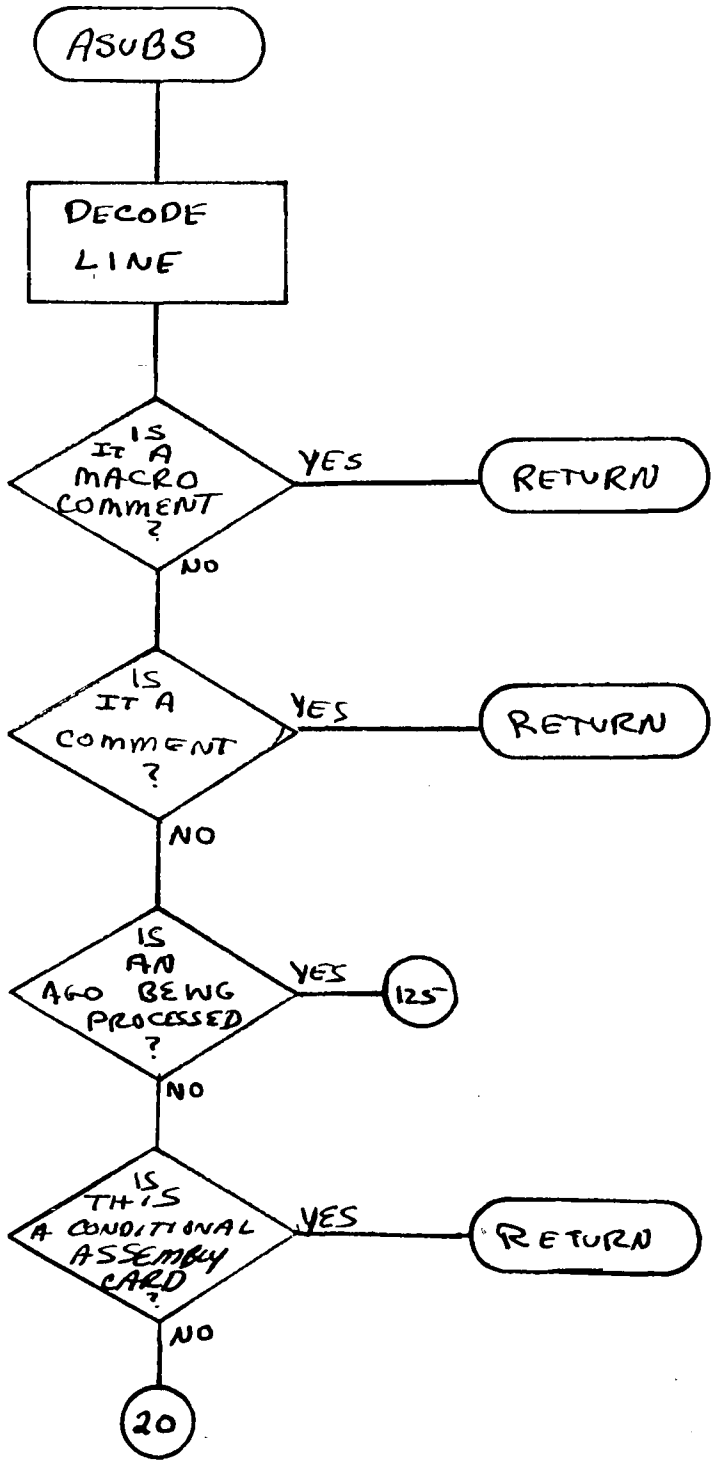

READ
MACRO CALL ?
YES
NO
READ FROM MDT
CALL ASUBS
CALL WORDS
RETURN
READ FROM INPUT
END OF FILE ?
YES
PRINT ERROR MESSAGE
EXIT
INCREMENT CARD COUNT
DECODE LINE
CALL WORDS
RETURN

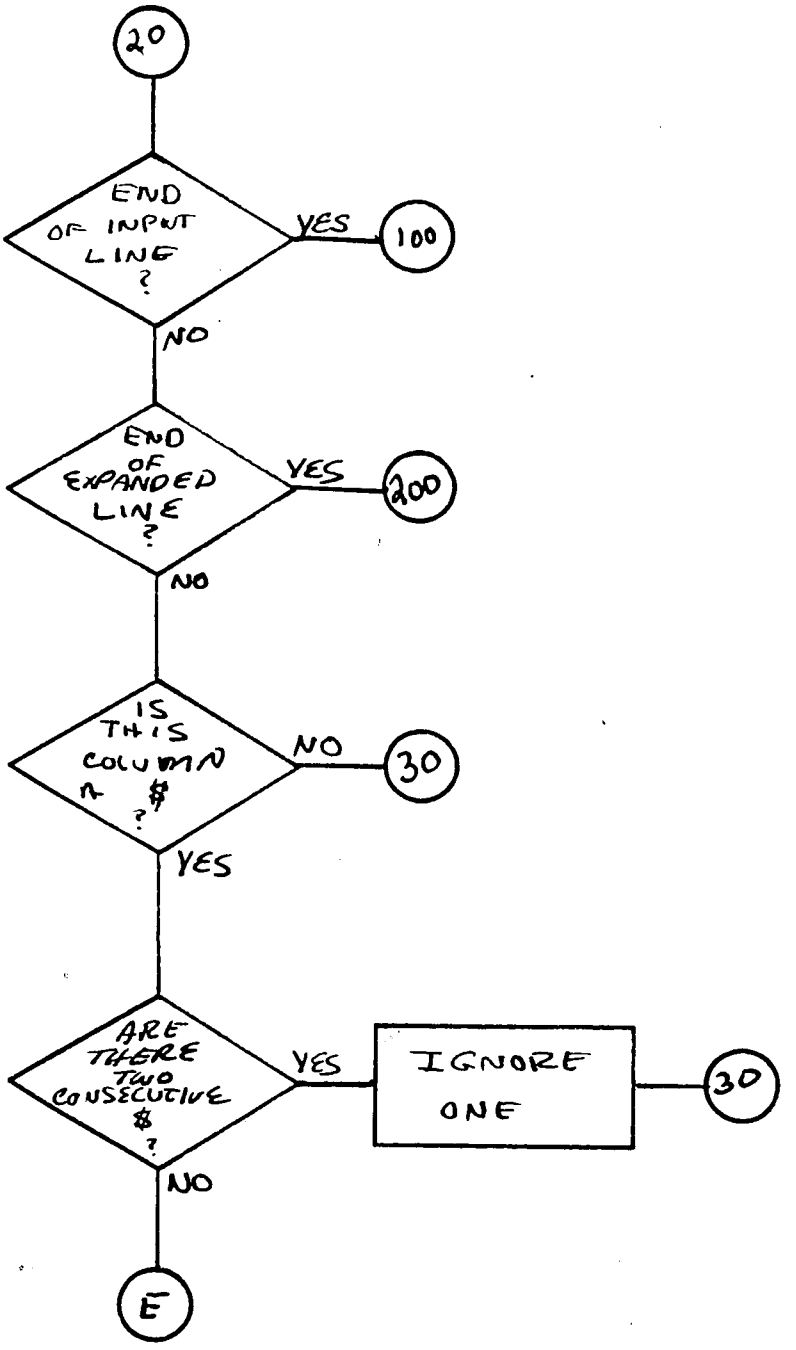
3.4.9 Subroutine ASUBS

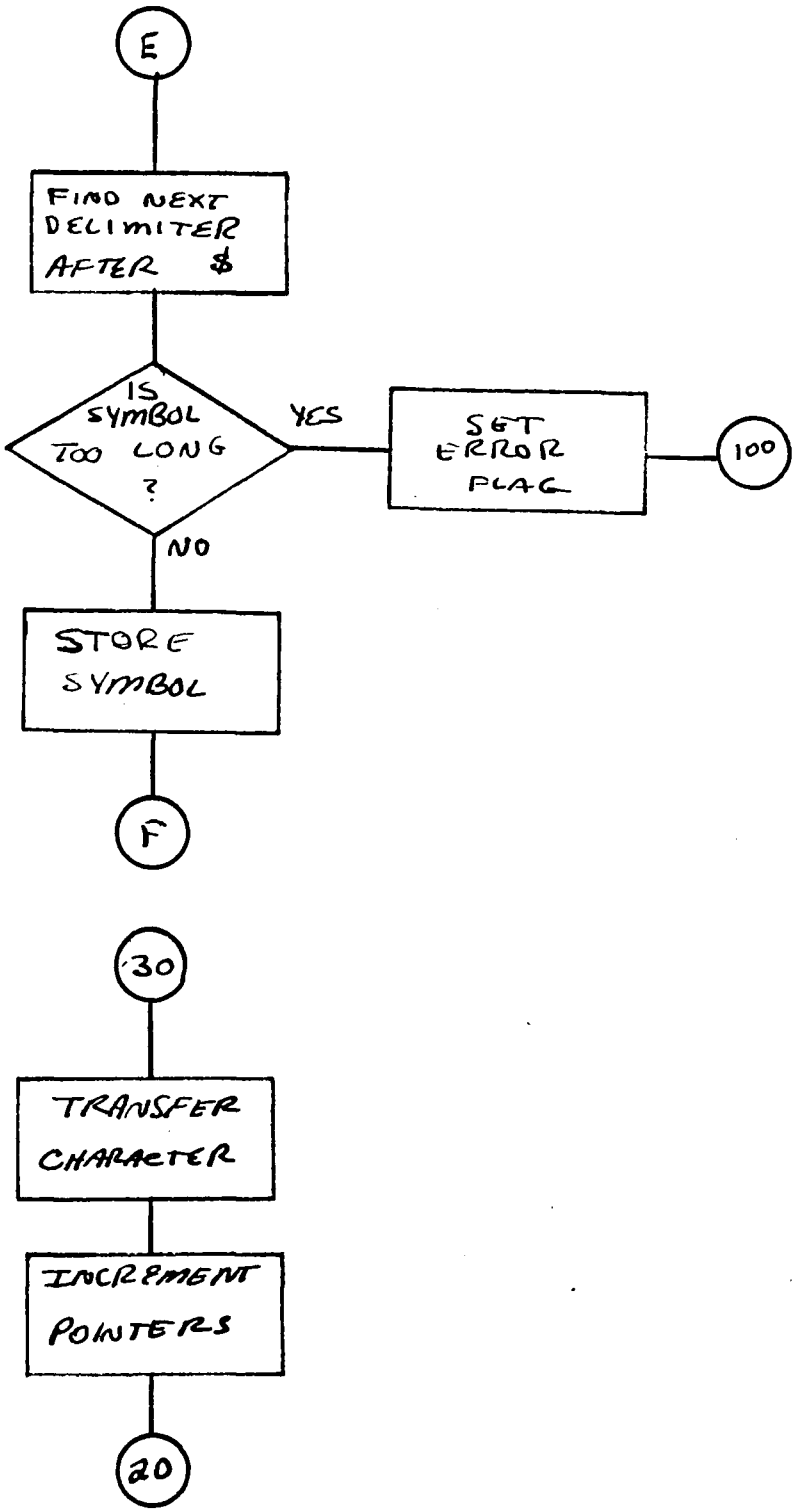
This Fortran routine replaces SET symbols with their current values and it separates sequence symbols from the input line. Certain cards are exempt from substitution or receive only limited processing. These include all comment cards, SET statements, and LCL and GBL statements. To obtain values for SET symbols, ASUBS calls REPSYM. The returned value is stored in the output array. ASUBS ensures that columns 72 through 80 are not destoyed if a line expands due to substitution. If any non-blank characters are lost due to substitution, an error message is printed.

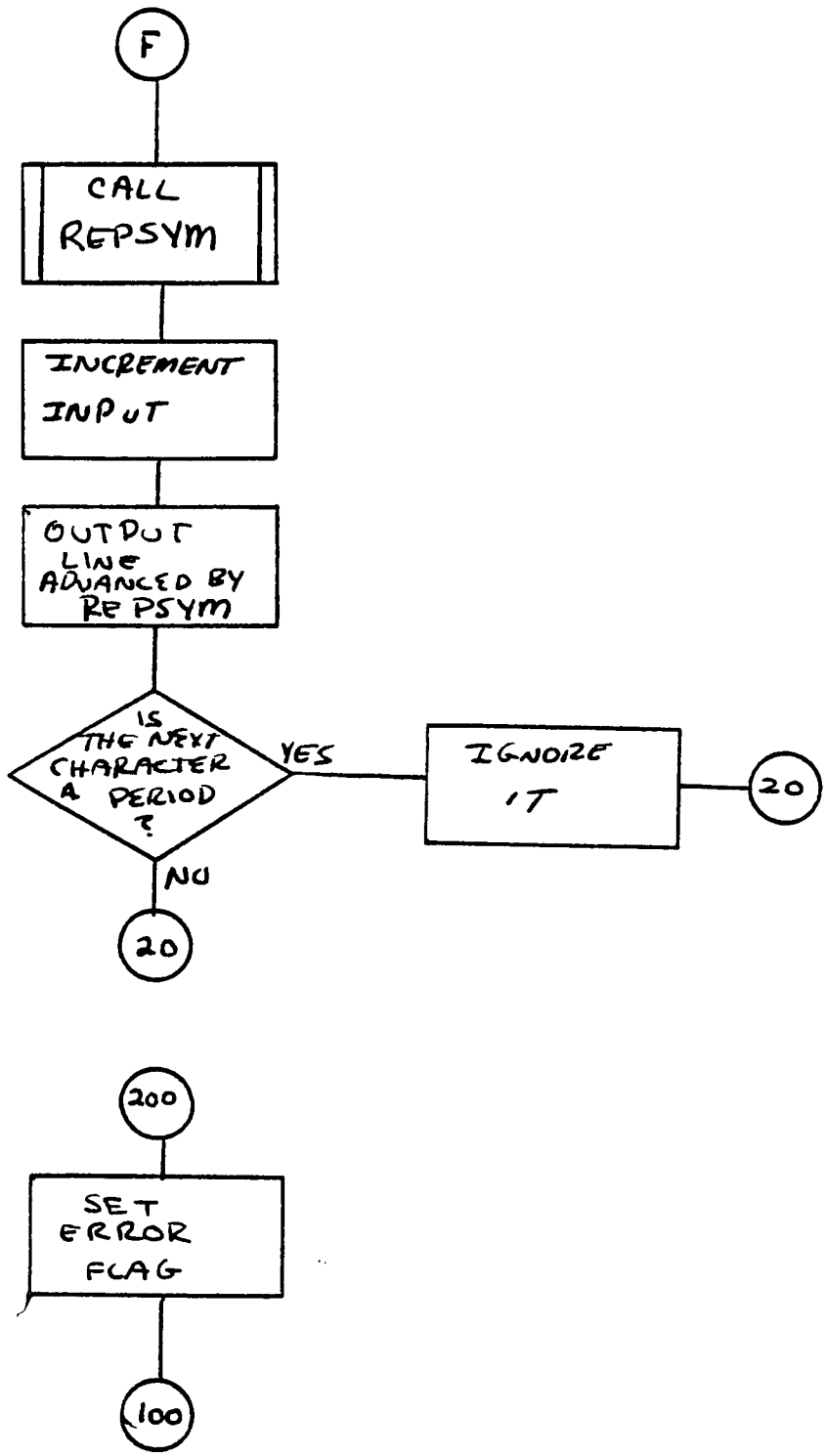
ASUBS first decodes the input line from 8A10 format to 80A1 format. It then processes the 80 character input array, IC1, one character at a time. Until a dollar sign is found, characters are transfered to the array IC2. If two consecutive dollar signs are found, only one is transfered. A single dollar sign is taken as the start of a SET symbol. The end of a SET symbol is indicated by any delimiter. REPSYM returns the SET symbol value in the next avilable words of IC2 and advances the pointer in that array. If the delimiter which terminated the symbol was a period, ASUBS recognizes it as the concatenation symbol and does not transfer it to the output array. The procedure continues, character by character, until either the input array is exhausted or the output array is filled.

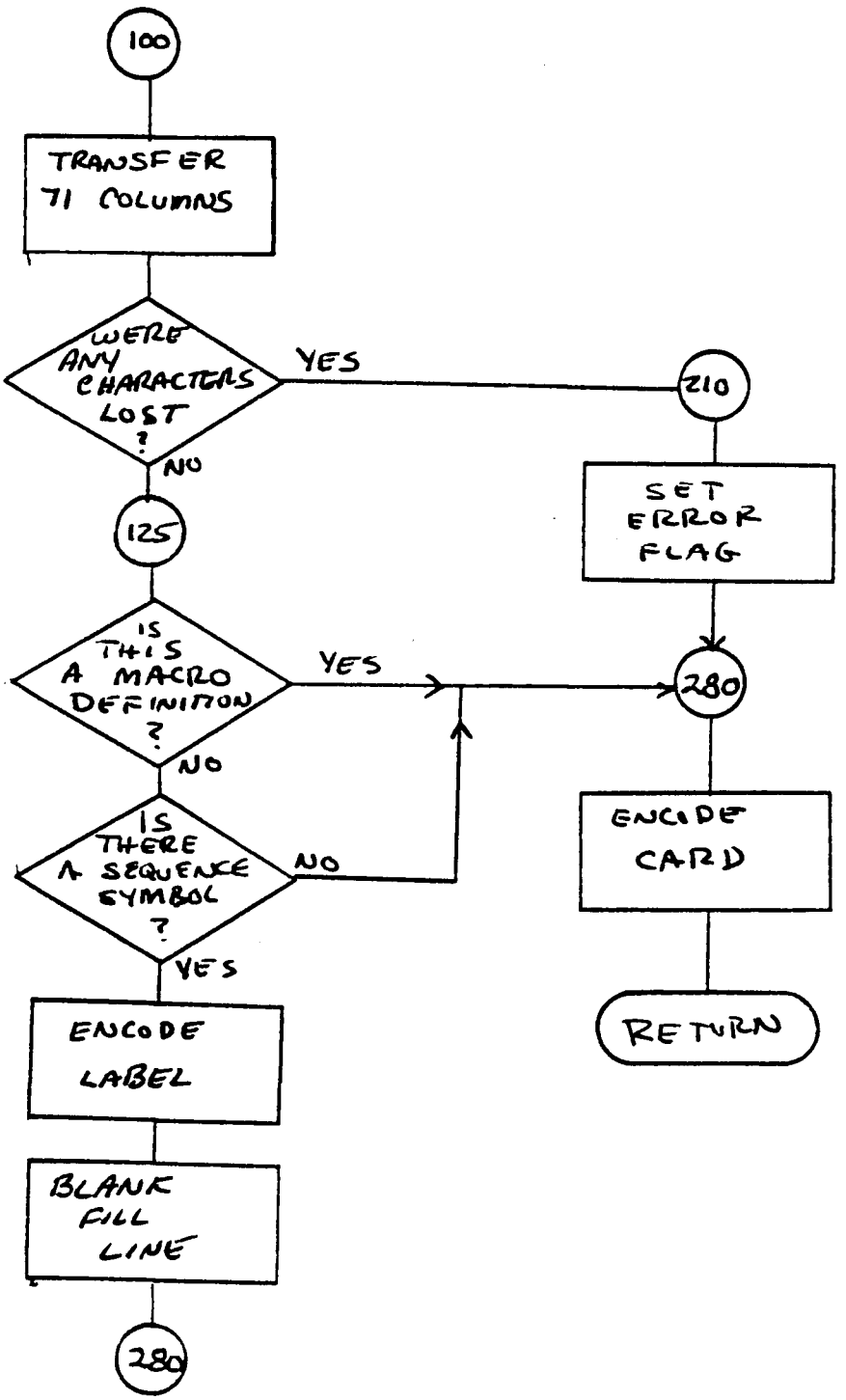
The first 71 characters of the expanded line are then transfered back to the input array, leaving columns 72 to 80 of the input array unchanged. If column one of the resulting array is a period, the sequence symbol is removed from the line and stored as a single word in IFIND. The positions it occupied on the card are blank filled. The resultant array IC1 is then converted back to 8A10 format, so that both versions of the input line contain the same information.


ASUBS
DECODE LINE
IS IT A MACRO COMMENT ?
YES
RETURN
NO
IS IT A COMMENT ?
YES
RETURN
NO
IS AN AGO BEING PROCESSED ?
YES
125
NO
IS THIS A CONDITIONAL ASSEMBLY CARD ?
YES
RETURN
NO
20


20
END OF INPUT LINE ?
YES
100
NO
END OF EXPANDED LINE ?
YES
200
NO
IS THIS COLUMN ^ $?
NO
30
YES
ARE THERE TWO CONSECUTIVE $?
YES
IGNORE ONE
30
NO
E

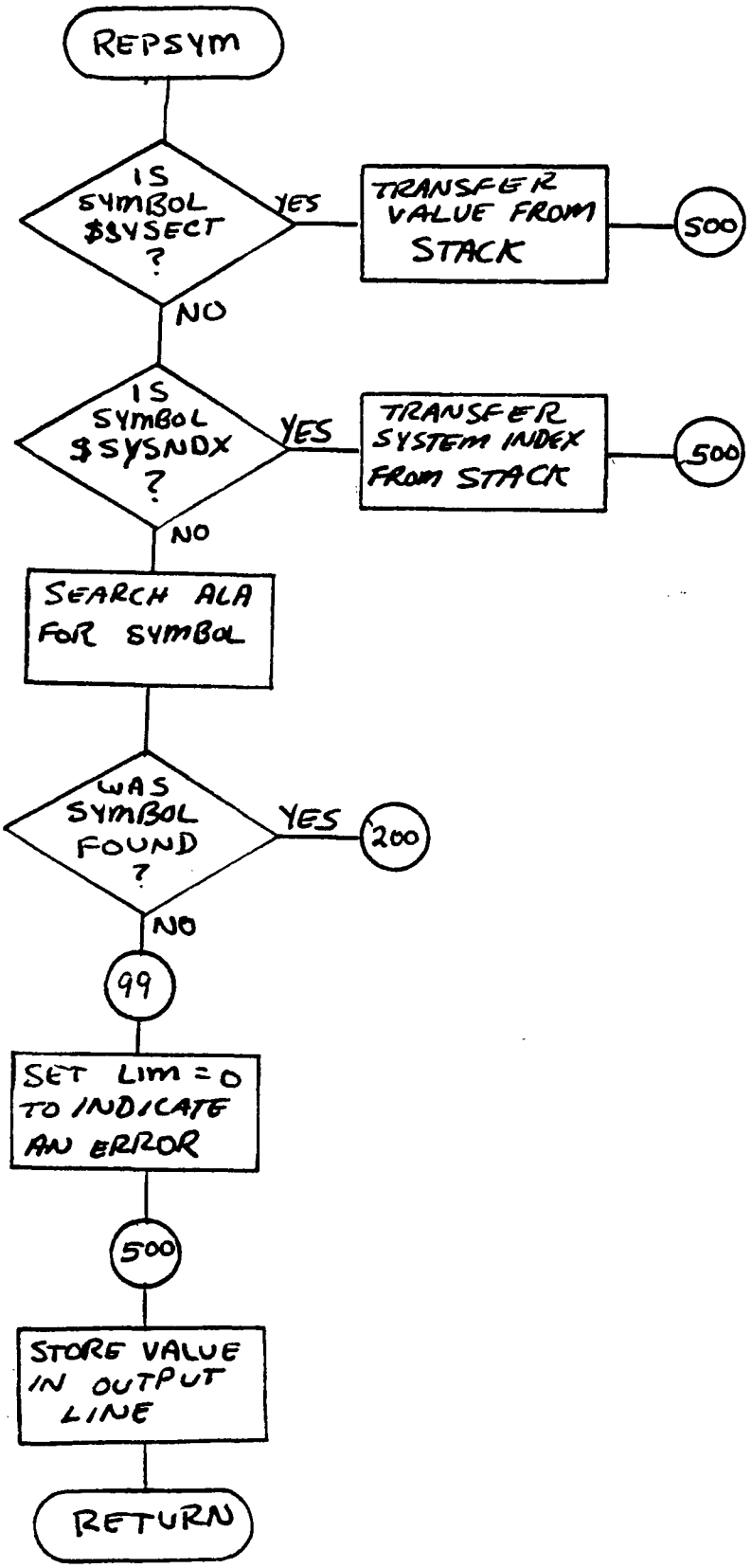

E
FIND NEXT DELIMITER AFTER $
IS SYMBOL TOO LONG ?
YES
SET ERROR FLAG
100
NO
STORE SYMBOL
F
30
TRANSFER CHARACTER
INCREMENT POINTERS
20

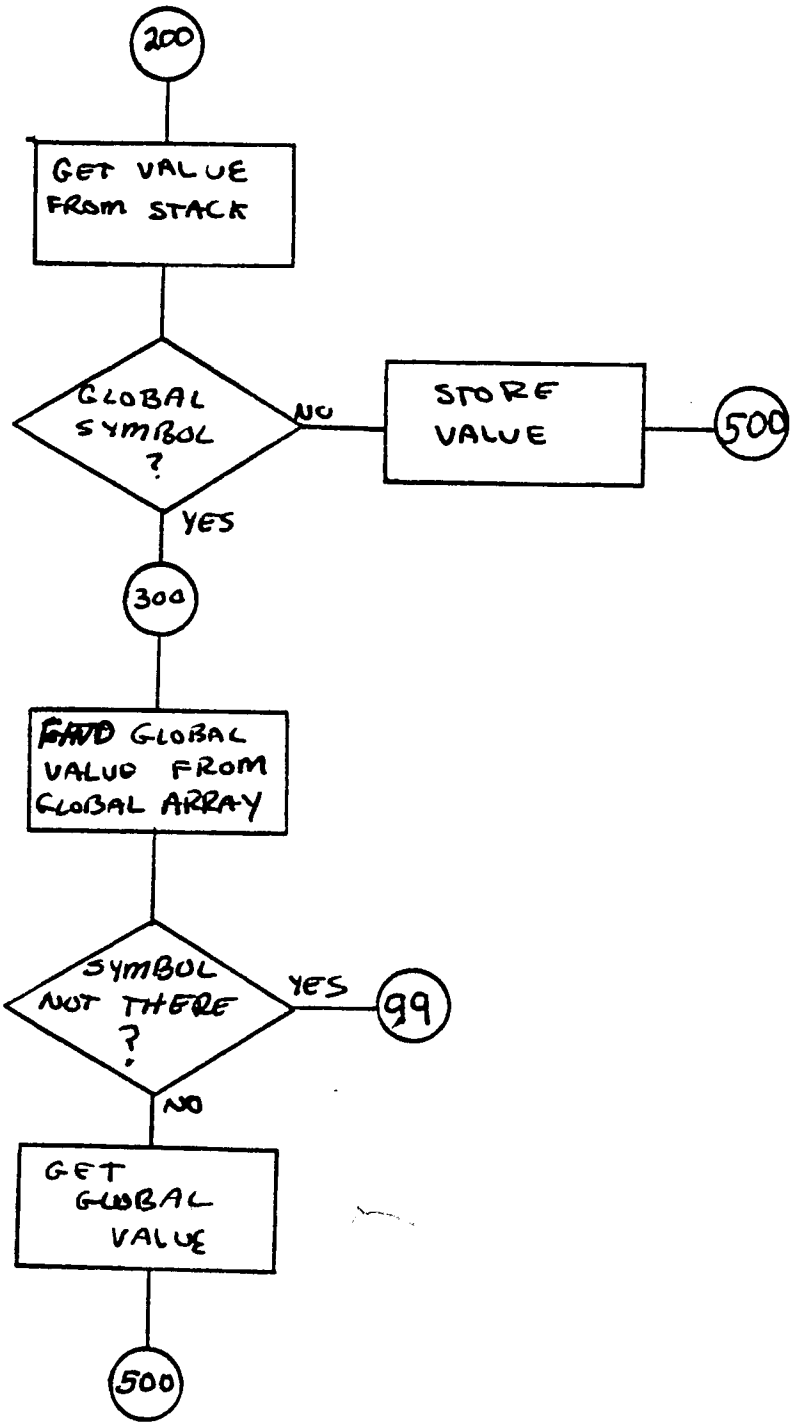

F
CALL REPSYM
INCREMENT INPUT
OUTPUT LINE ADVANCED BY REPSYM
IS THE NEXT CHARACTER A PERIOD ?
YES
IGNOIZE IT
20
NO
20
200
SET ERROR FLAG
100


100
TRANSFER 71 COLUMNS
WERE ANY CHARACTERS LOST ?
YES
210
SET ERROR FLAG
NO
125
IS THIS A MACRO DEFINITION ?
YES
280
NO
ENCODE CARD
IS THERE A SEQUENCE SYMBOL ?
NO
YES
RETURN
ENCODE LABEL
BLANK FILL LINE
280

3.4.10 Subroutine REPSYM

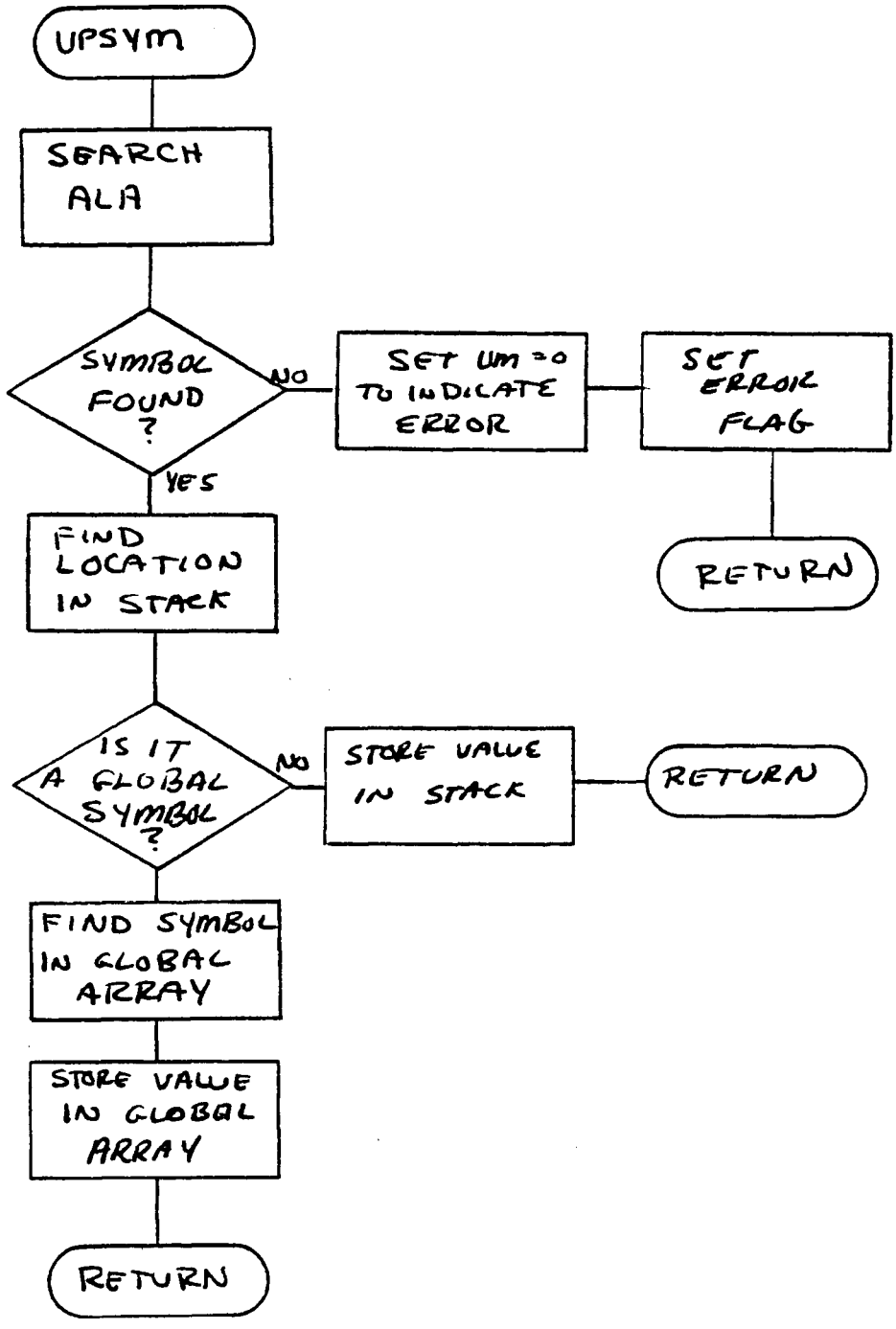
This Fortran subroutine looks up the current value of a SET symbol as found by ASUBS or in a SET statement operand. The value is returned in an array of characters in a format designed particularly for ASUBS, but easily used by other routines. REPSYM removes the value from the stack for most symbols. If the symbol is global, its value is obtained from the global array; if it is a system variable its value is obtained from the stack in a special manner.


REPSYM
IS SYMBOL $SYSECT ?
YES
TRANSFER VALUE FROM STACK
500
NO
IS SYMBOL $SYSNDX ?
YES
TRANSFER SYSTEM INDEX FROM STACK
500
NO
SEARCH ALA FOR SYMBOL
WAS SYMBOL FOUND ?
YES
200
NO
99
SET LIM = 0 TO INDICATE AN ERROR
500
STORE VALUE IN OUTPUT LINE
RETURN


200
GET VALUE FROM STACK
GLOBAL SYMBOL ?
NO
STORE VALUE
500
YES
300
FIND GLOBAL VALUE FROM GLOBAL ARRAY
SYMBOL NOT THERE ?
YES
99
NO
GET GLOBAL VALUE
500

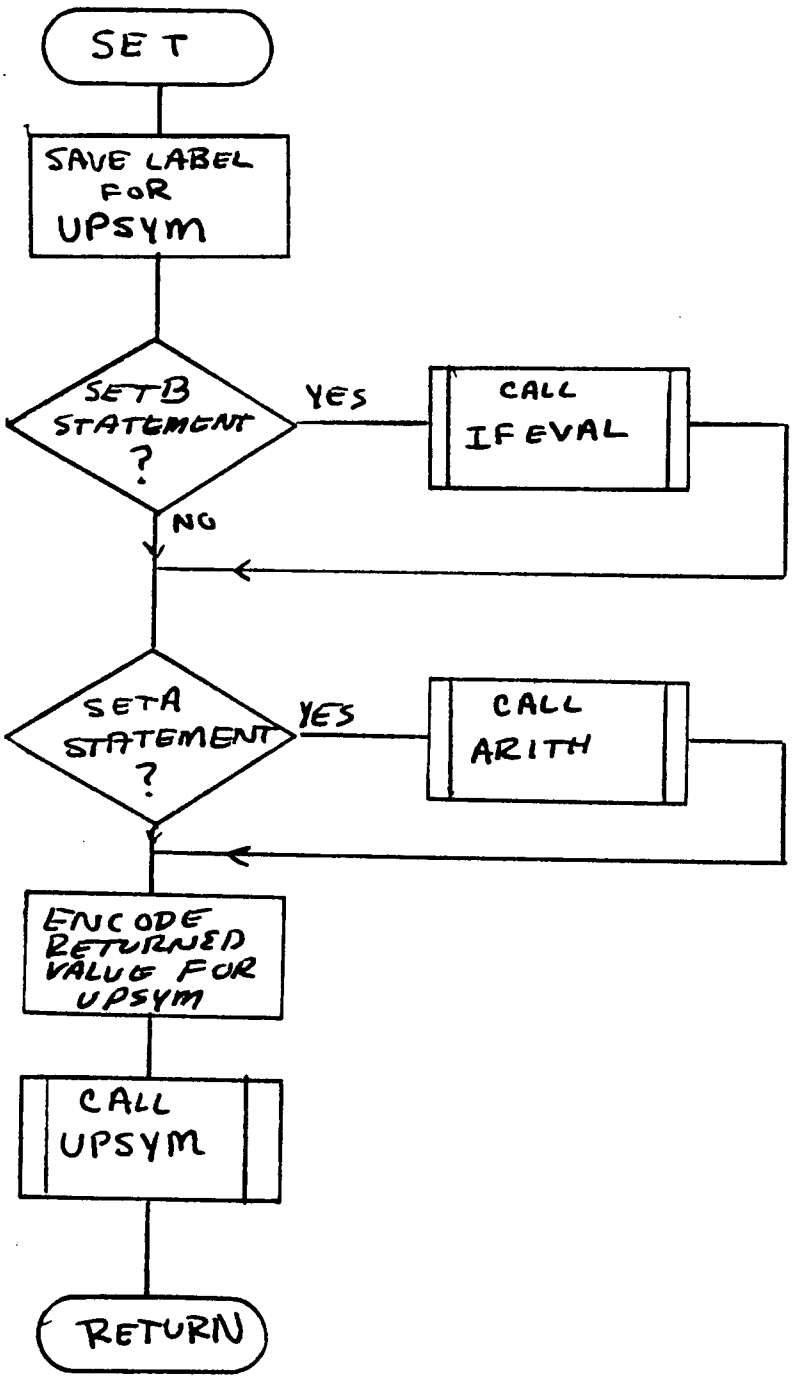
3.4.10 Subroutine UPSYM

This Fortran subroutine stores the value of a SET symbol. The value is stored in the stack or in the global array. The subroutine is essentially the inverse function of subroutine REPSYM. It is called when a SET symbol is evaluated. No special processing need be done for system variables because their values may not be changed by the user.


UPSYM
SEARCH ALA
SYMBOL FOUND ?
NO
SET UM = 0 TO INDICATE ERROR
SET ERROR FLAG
RETURN
YES
FIND LOCATION IN STACK
IS IT A GLOBAL SYMBOL ?
NO
STORE VALUE IN STACK
RETURN
FIND SYMBOL IN GLOBAL ARRAY
STORE VALUE IN GLOBAL ARRAY
RETURN

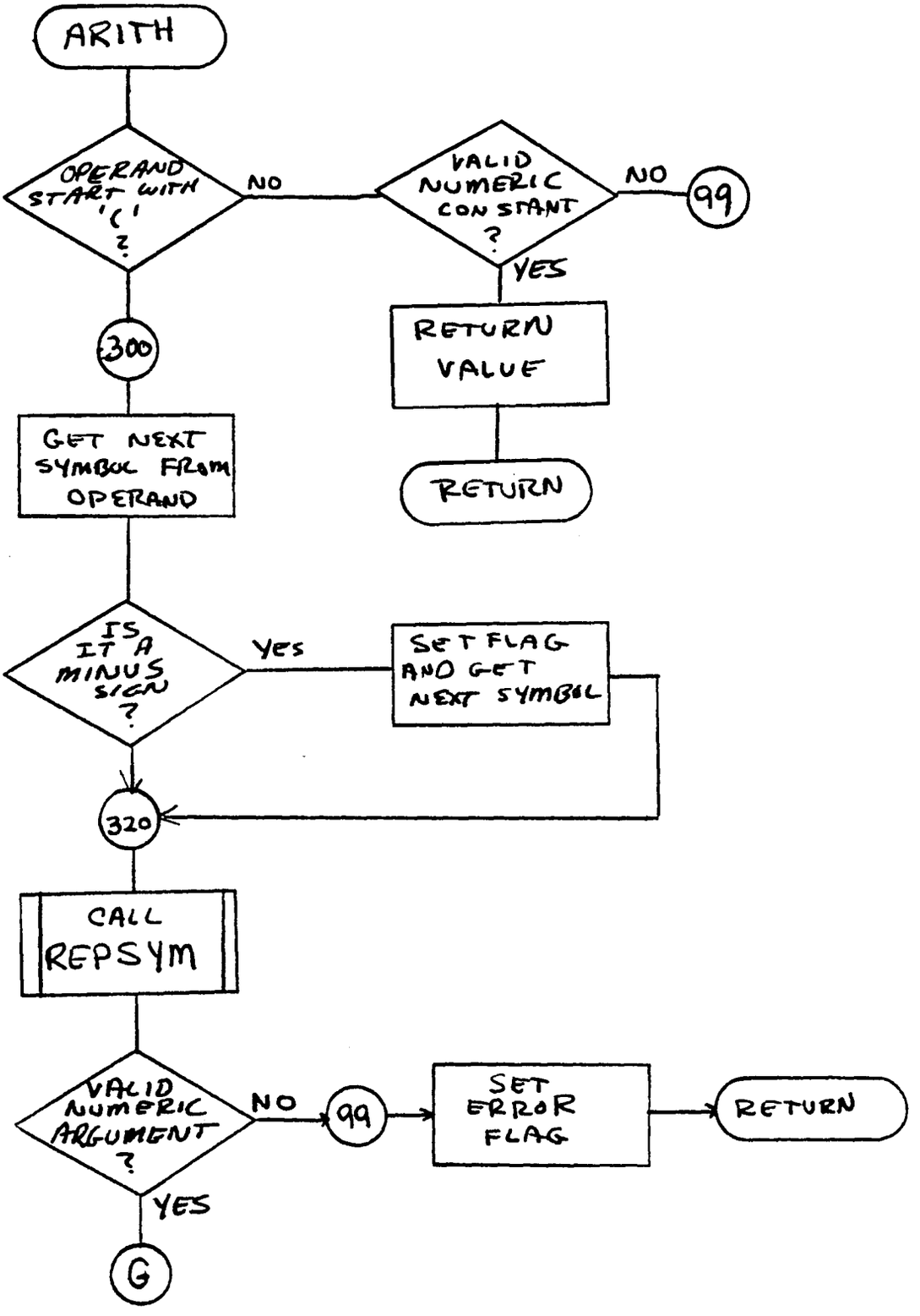
34.12 Subroutine SET

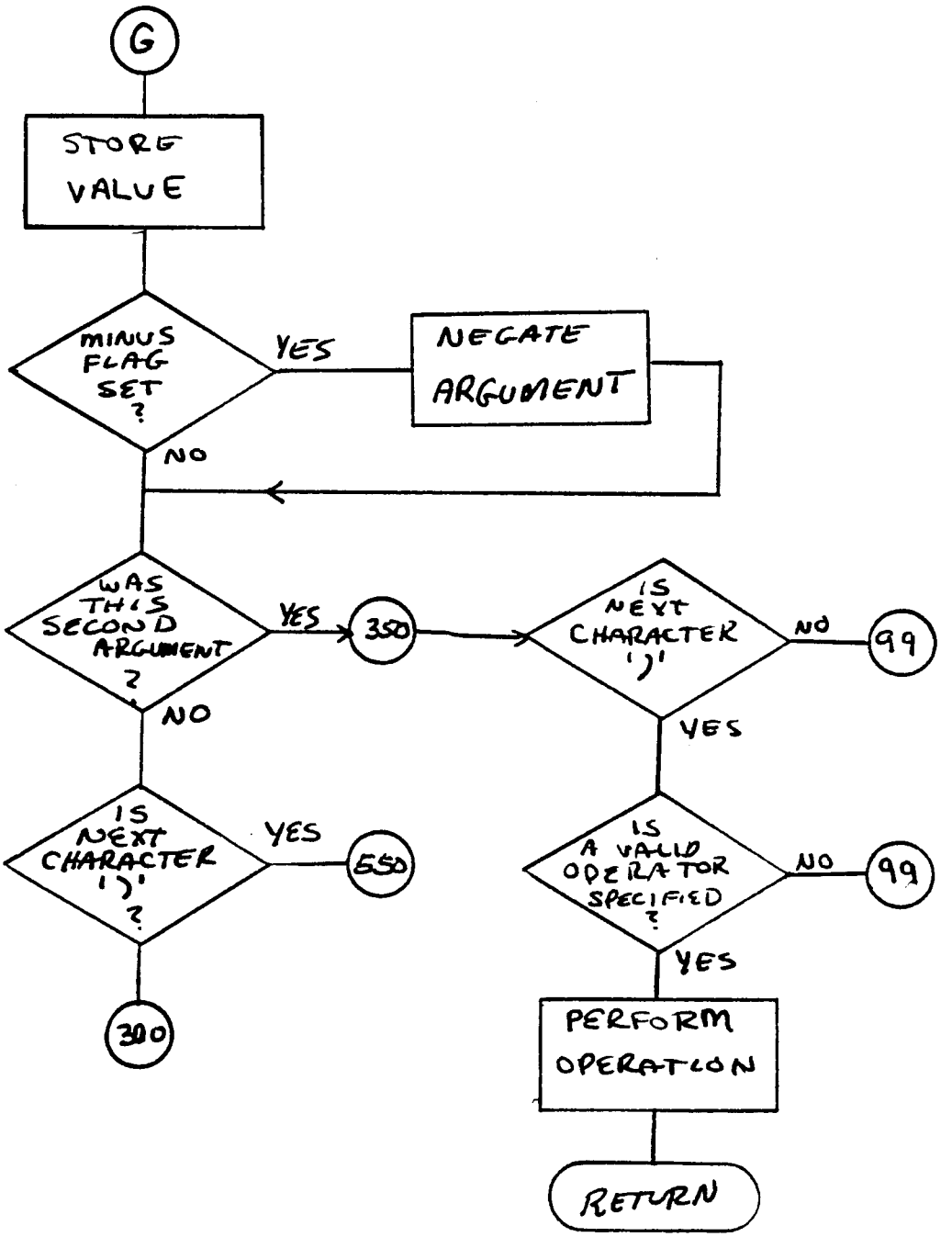
This Fortran routine is called by ASSEMBL to evaluate SETA and SETB statements. It stores the symbol to be replaced and then calls either ARITH or IFEVAL to evaluate the operand of the SETA or SETB statement. The value returned by these two routines is passed to UPSYM along with the name from the label field. UPSYM stores the value in the stack or global array.


SET
SAVE LABEL FOR UPSYM
SETB STATEMENT ?
YES
CALL IFEVAL
NO
SETA STATEMENT ?
YES
CALL ARITH
ENCODE RETURNED VALUE FOR UPSYM
CALL UPSYM
RETURN

3.4.13 Subroutine ARITH

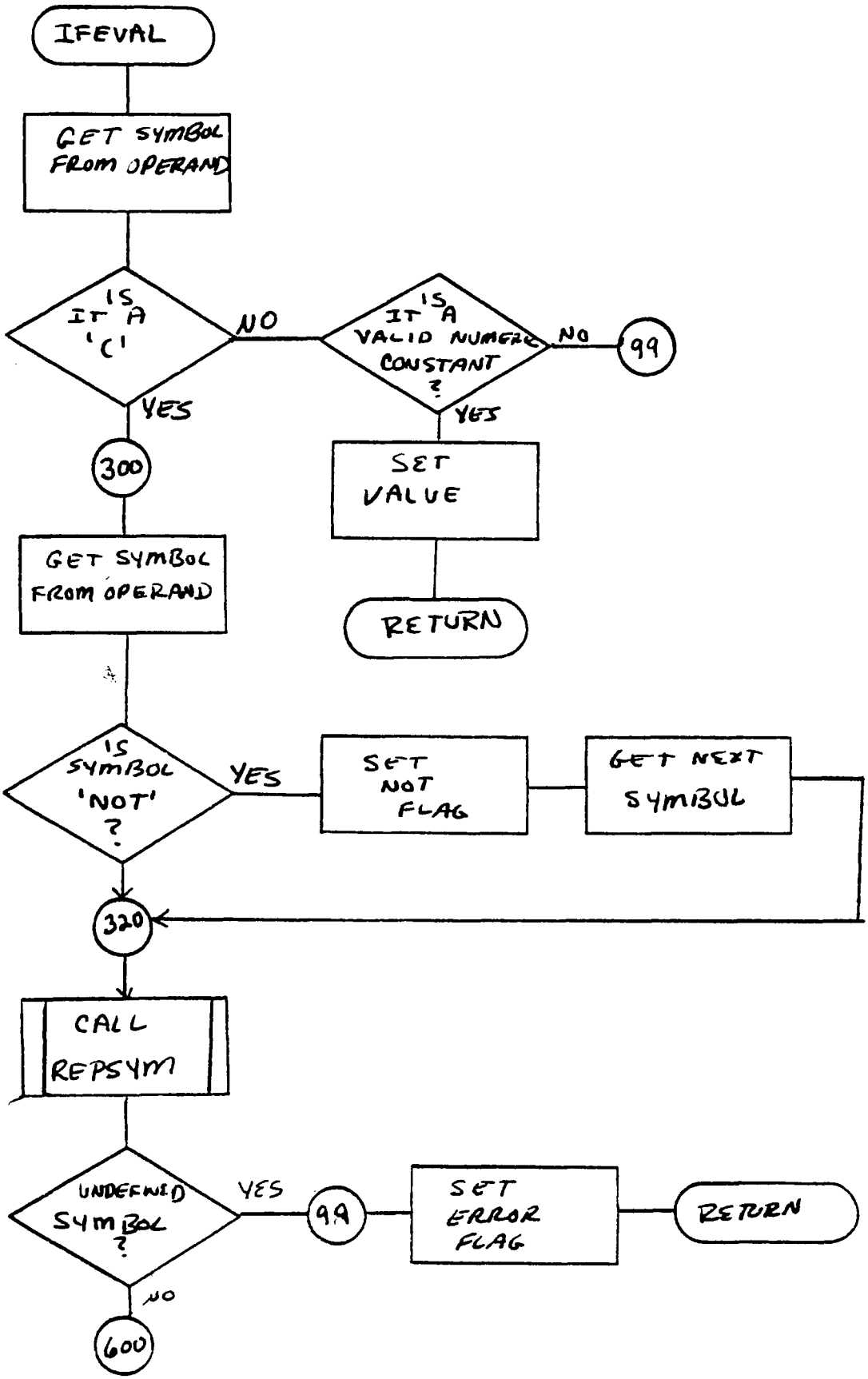
This Fortran subroutine evaluates the operand of a SETA statement. It returns an integer value which is the result of the arithmetic operations given by the operand. It calls REPSYM to get the value for each SET symbol encountered in the operand and WORDS to get symbols from the card image.

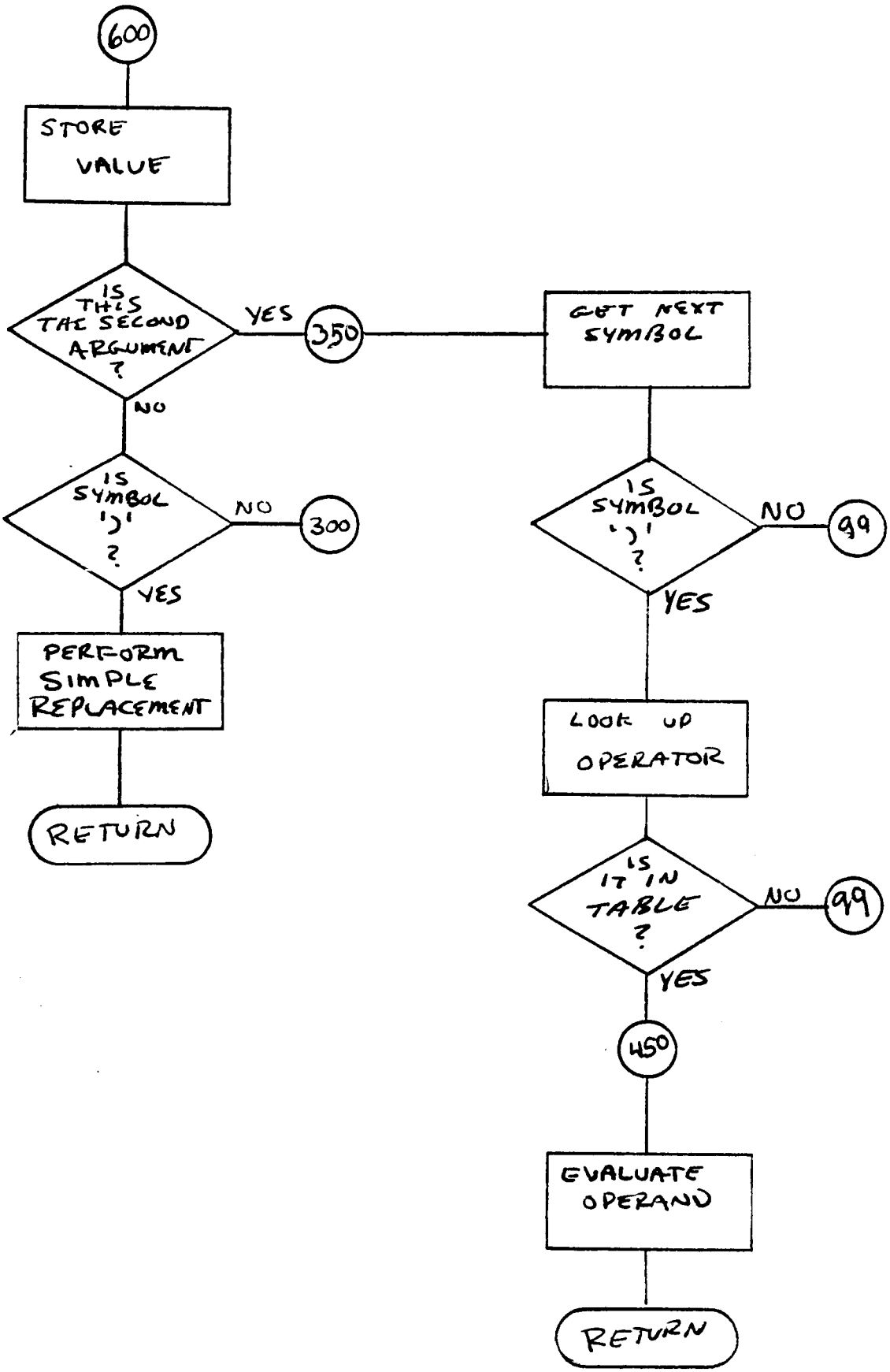

ARITH
OPERAND START WITH '('?
NO
VALID NUMERIC CONSTANT?
NO
99
YES
RETURN VALUE
RETURN
300
GET NEXT SYMBOL FROM OPERAND
IS IT A MINUS SIGN?
Yes
SET FLAG AND GET NEXT SYMBOL
320
CALL REPSYM
VALID NUMERIC ARGUMENT?
NO
99
SET ERROR FLAG
RETURN
YES
G


G
STORE VALUE
MINUS FLAG SET ?
YES
NEGATE ARGUMENT
NO
WAS THIS SECOND ARGUMENT ?
YES
350
IS NEXT CHARACTER ')'
NO
99
YES
NO
IS NEXT CHARACTER ')' ?
YES
550
IS A VALID OPERATOR SPECIFIED ?
NO
99
YES
300
PERFORM OPERATION
RETURN

3.4.14 Subroutine IFEVAL

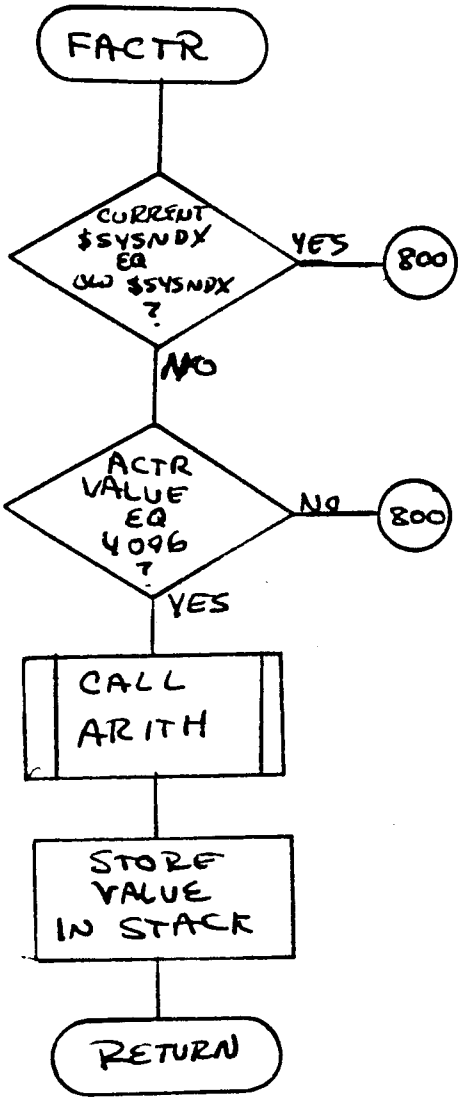
This Fortran subroutine evaluates the operand of SETB and AIF statements. It returns a numerical value of one if the expression is true and zero if it is false. It calls REPSYM to get values for SET symbols and it calls WORDS to get symbols from the card image. If logical operators are used in the expression, the values of the SET symbols in the expression must be one or zero and NOT operators may be included. If relational operators are used, the SET symbols may take on any value. If the SET symbol value is purely numeric as determined by subroutine NUMBCHK, the value is stored as an integer, otherwise it is stored in display code. Character strings are stored, in this routine only, with the length in the upper six bits of a computer word. Consequently, negative values will never occur as they could if a character was stored in the upper bits. Also, when two strings are compared, the longer will always be greater. A side effect of this format is that character values always compare greater than a numeric value.


IFEVAL
GET SYMBOL FROM OPERAND
IS IT A 'C'
NO
IS IT A VALID NUMERIC CONSTANT ?
NO
99
YES
YES
300
SET VALUE
GET SYMBOL FROM OPERAND
RETURN
IS SYMBOL 'NOT' ?
YES
SET NOT FLAG
GET NEXT SYMBOL
320
CALL REPSYM
UNDEFINED SYMBOL ?
YES
99
SET ERROR FLAG
RETURN
NO
600


600
STORE VALUE
IS THIS THE SECOND ARGUMENT ?
YES
350
NO
IS SYMBOL ')' ?
NO
300
YES
PERFORM SIMPLE REPLACEMENT
RETURN
GET NEXT SYMBOL
IS SYMBOL ')' ?
NO
99
YES
LOOK UP OPERATOR
IS IT IN TABLE ?
NO
99
YES
450
EVALUATE OPERAND
RETURN

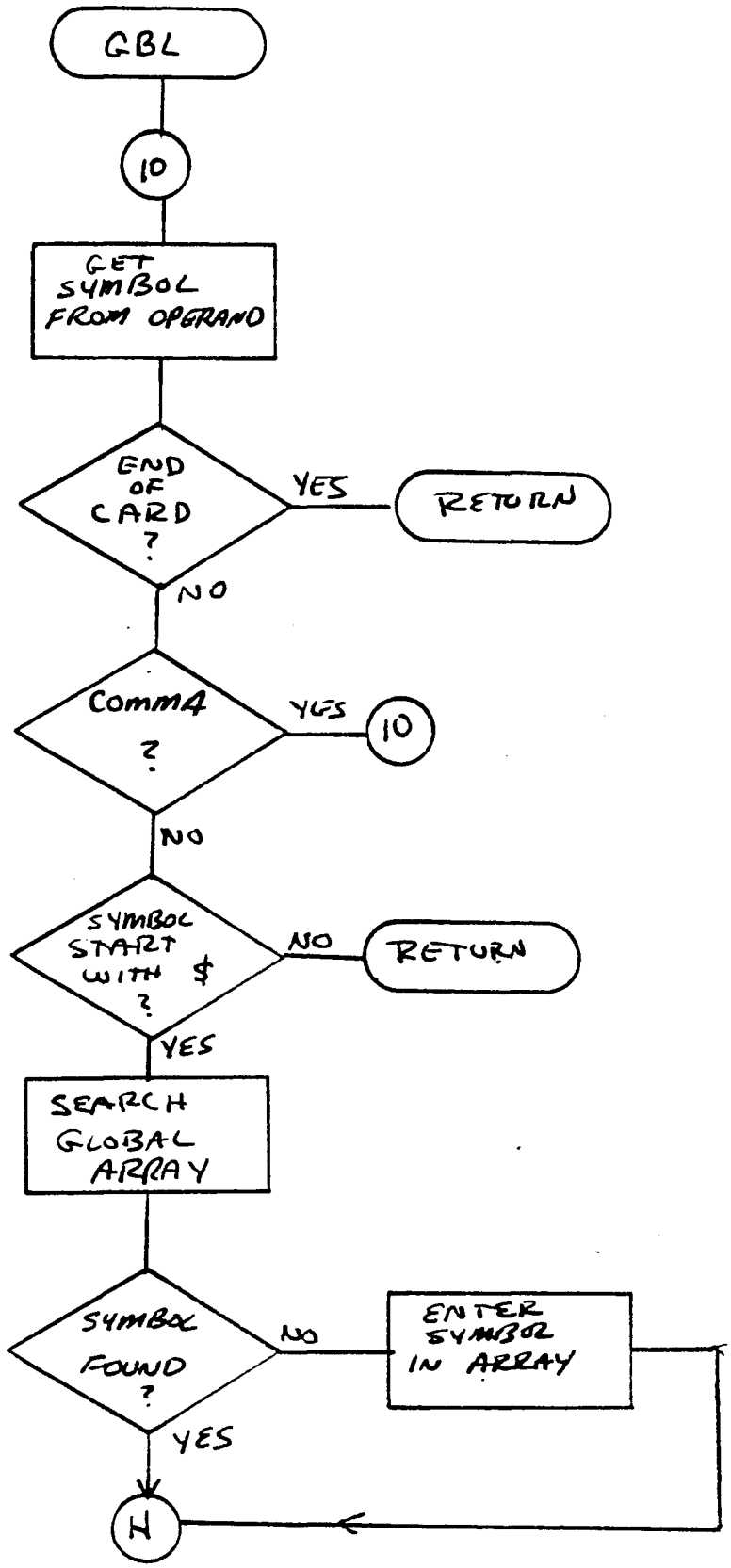
3.4.15 Subroutine FACTR

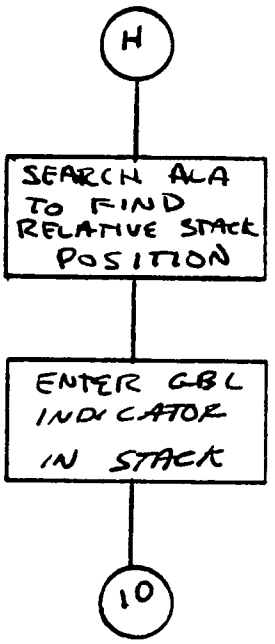
This Fortran subroutine is called when an ACTR statement is detected by ASSEMBL. It, in turn, calls ARITH to evaluate its operand and stores the returned value in the third word of the stack. In order to prevent multiple ACTR statements within a macro loop, this routine performs two tests. The first ensures that two consecutive ACTR statements did not occur in the same macro. The second ensures that the old ACTR value is still set at its default value of 4096.


FACTR
CURRENT $SYSNDX EQ OLD $SYSNDX ?
YES
800
NO
ACTR VALUE EQ 4096 ?
NO
800
YES
CALL ARITH
STORE VALUE IN STACK
RETURN

3.4.16 Subroutine GBL

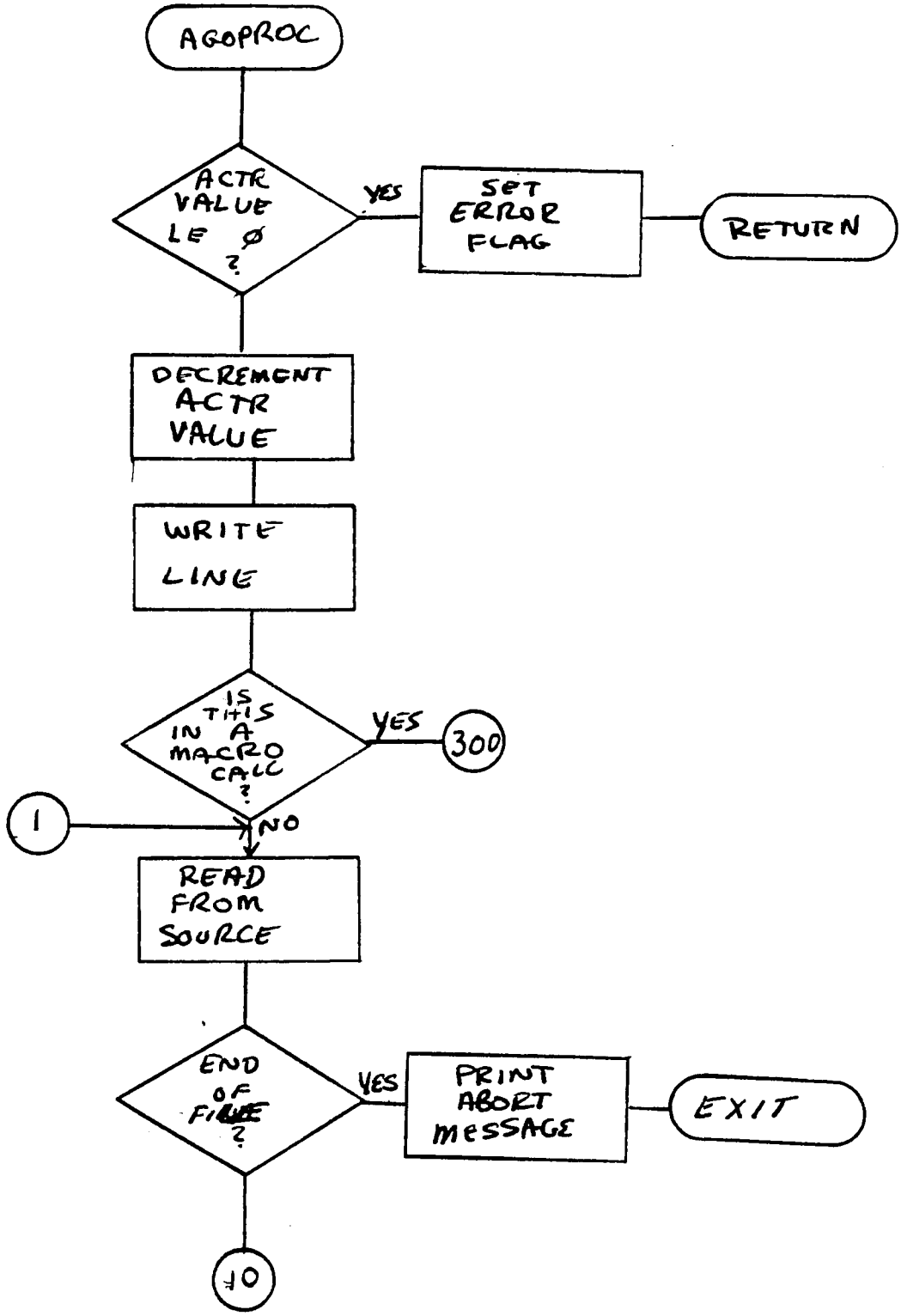
This Fortran routine is called by ASSEMBL when GBL cards are processed. Subroutine ARGUMNT has already been called to enter the values into the stack by the time that GBL is called. GBL enters the global indicator into the stack to distinguish globals from local variables. This indicator is currently Hollerith constant '$$GBL,,$$9'. As long as there is no character SET pseudo-op there will be no way for the user to enter this value into the stack. After marking the stack entry, GBL searches the global array in common block /GLOBALS/. If the global is not already defined there, it is entered at this time.

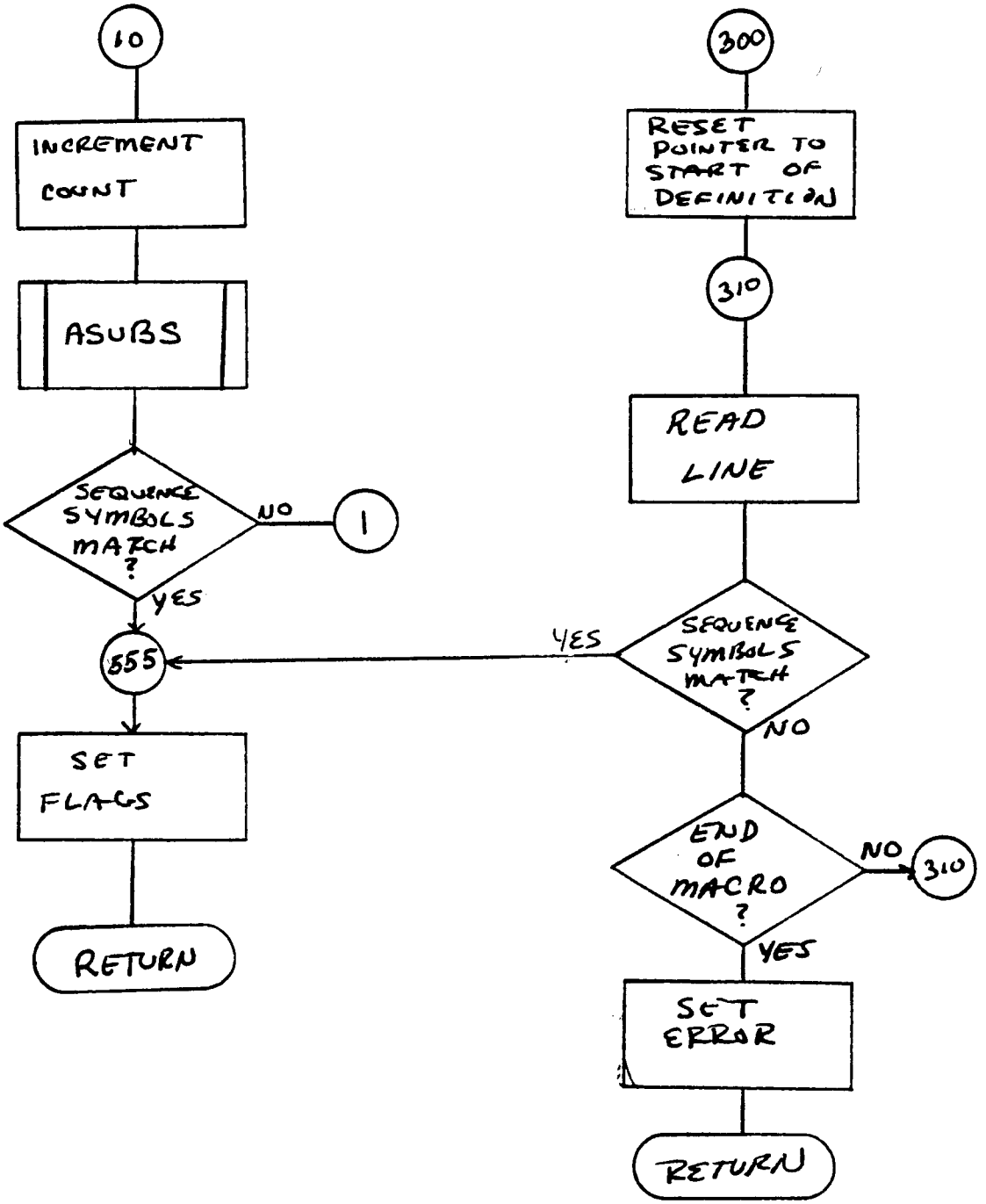

GBL
10
GET SYMBOL FROM OPGRAND
END OF CARD ?
YES
RETURN
NO
COMMA ?
YES
10
NO
SYMBOL START WITH $?
NO
RETURN
YES
SEARCH GLOBAL ARRAY
SYMBOL FOUND ?
NO
ENTER SYMBOL IN ARRAY
YES
H


H
SEARCH ALA TO FIND RELATIVE STACK POSITION
ENTER GBL INDICATOR IN STACK
10

34.17 Subroutine AGOPROC

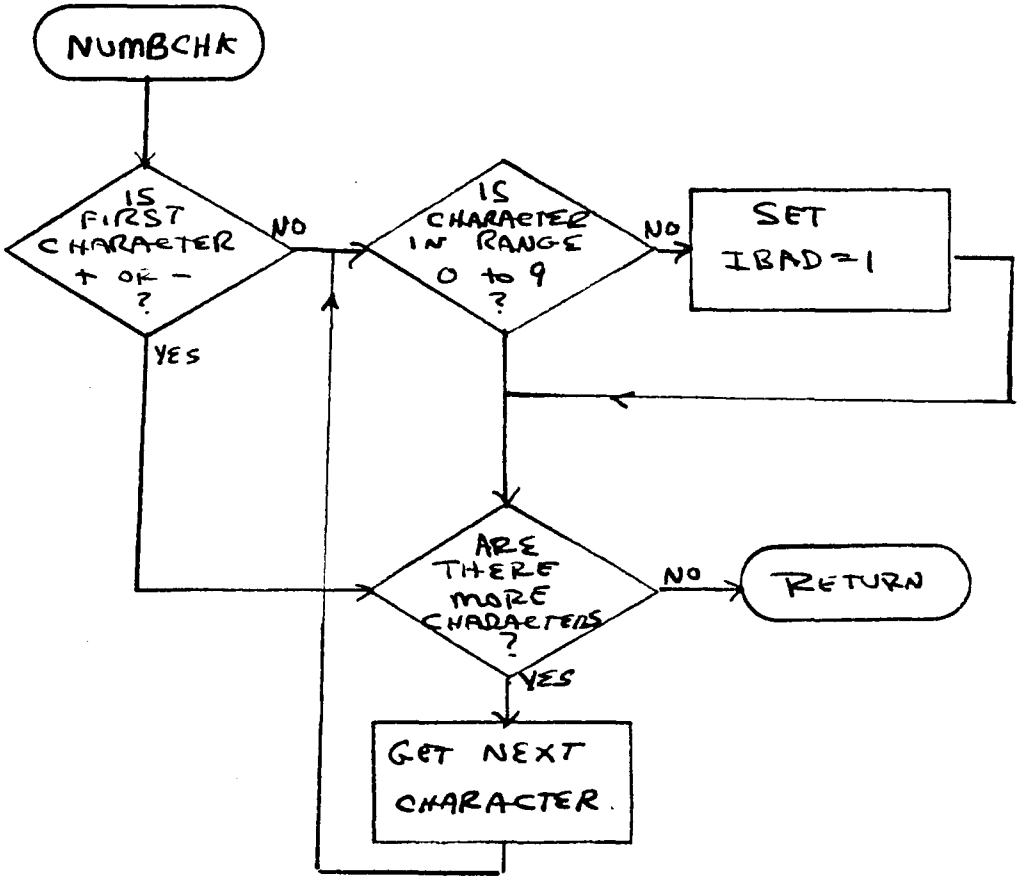
This Fortran subroutine is called by ASSEMBL when an AGO or a true AIF statement is encountered. It reads from either the INPUT file or the MDT file until it finds the specified sequence symbol. If the symbol is not found when the end of the macro definition is reached, an error is printed and control is returned. If end of file is reached when reading from the INPUT file, a message is printed and the job is aborted. This routine calls subroutine ASUBS to get the sequence symbol from a card, however no other processing is performed by ASUBS at this time.


AGOPROC
ACTR VALUE LE Ø ?
YES
SET ERROR FLAG
RETURN
DECREMENT ACTR VALUE
WRITE LINE
IS THIS IN A MACRO CALL ?
YES
300
1
NO
READ FROM SOURCE
END OF FILE ?
YES
PRINT ABORT MESSAGE
EXIT
10


10
INCREMENT COUNT
ASUBS
SEQUENCE SYMBOLS MATCH ?
NO
1
YES
555
SET FLAGS
RETURN
300
RESET POINTER TO START OF DEFINITION
310
READ LINE
SEQUENCE SYMBOLS MATCH ?
YES
NO
END OF MACRO ?
NO
310
YES
SET ERROR
RETURN

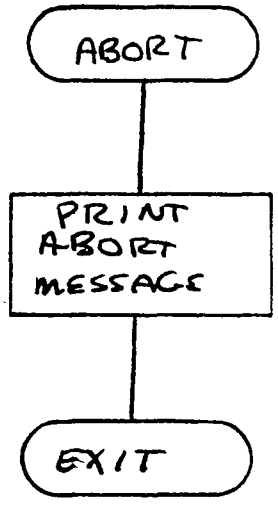
3.4.18 Subroutine NUMBCHK

This Fortran routine checks that the digits passed to it are valid numeric digits. The first word of the array to be checked and the number of digits in it are passed as parameters. A third parameter is used as a flag to indicate the result of the test. A plus or minus sign may be the first character in the array, however the remaining characters must all be numeric. The purpose of this routine is to ensure that a format error will not occur on an ENCODE statement.


NUMBCHK
IS FIRST CHARACTER + OR - ?
NO
YES
IS CHARACTER IN RANGE 0 to 9 ?
NO
SET IBAD=1
ARE THERE MORE CHARACTERS ?
NO
RETURN
YES
GET NEXT CHARACTER.

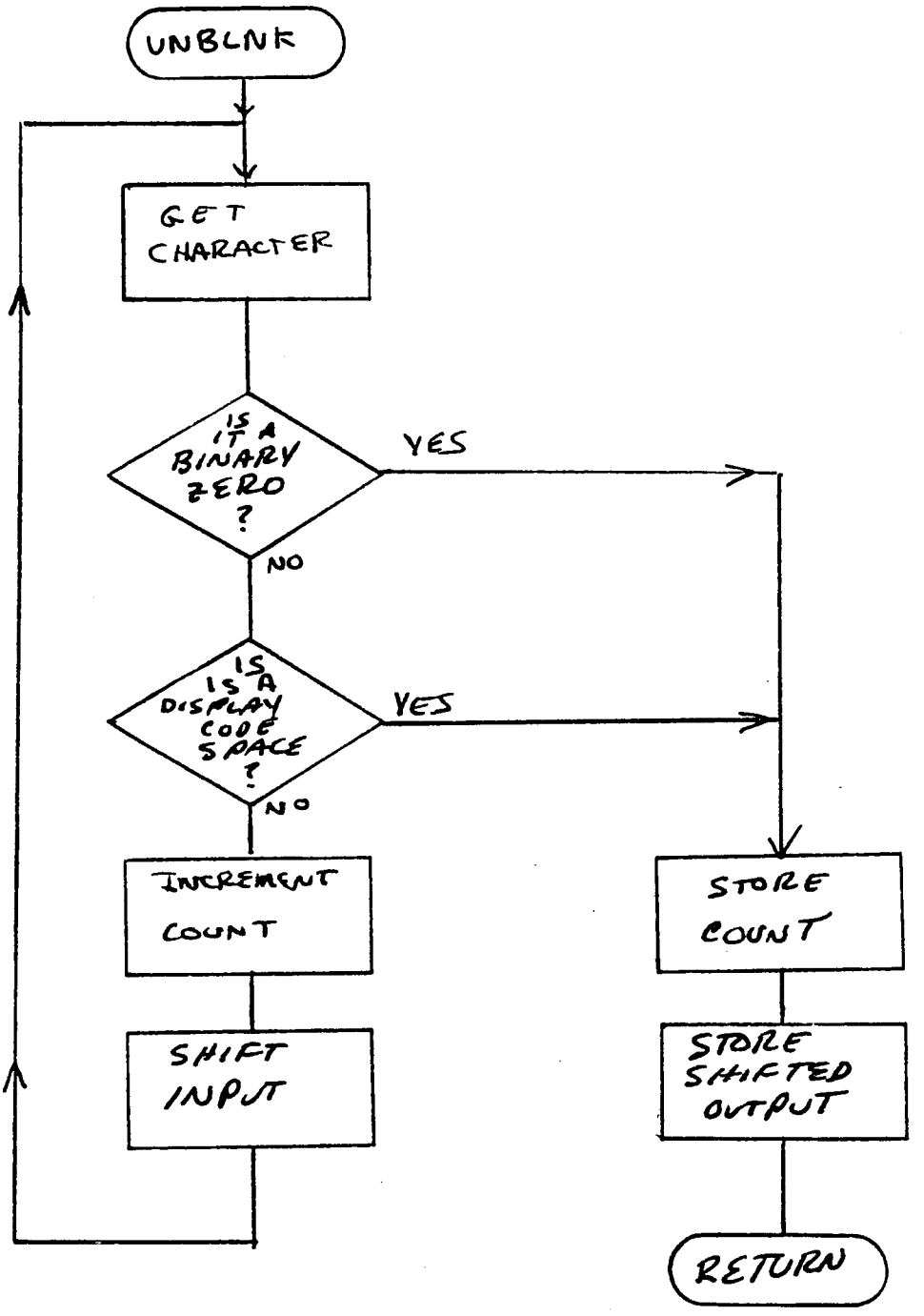
3.4.19 Subroutine ABORT

This Fortran subroutine is called whenever a macro table overflows during first pass. It prints one of four error messages and then terminates the program.



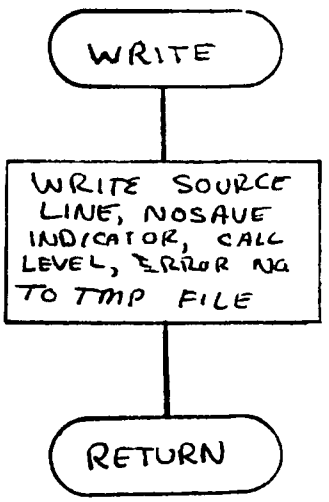
3.4.20 Subroutine UNBLNK

This is a COMPASS subroutine which accepts a right justified symbol and returns a left justified word. It also counts the number of non-blank characters in the word. This routine is used primarily for converting numbers from I10 format into a character string with length as stored in the stack. It begins processing from the righthand end of a word terminating when it encounters a blank or a zero byte.


UNBLNK
GET CHARACTER
IS IT A BINARY ZERO ?
YES
NO
IS IS A DISPLAY CODE SPACE ?
YES
NO
INCREMENT COUNT
SHIFT INPUT
STORE COUNT
STORE SHIFTED OUTPUT
RETURN

3.4.21 Subroutine WRITE

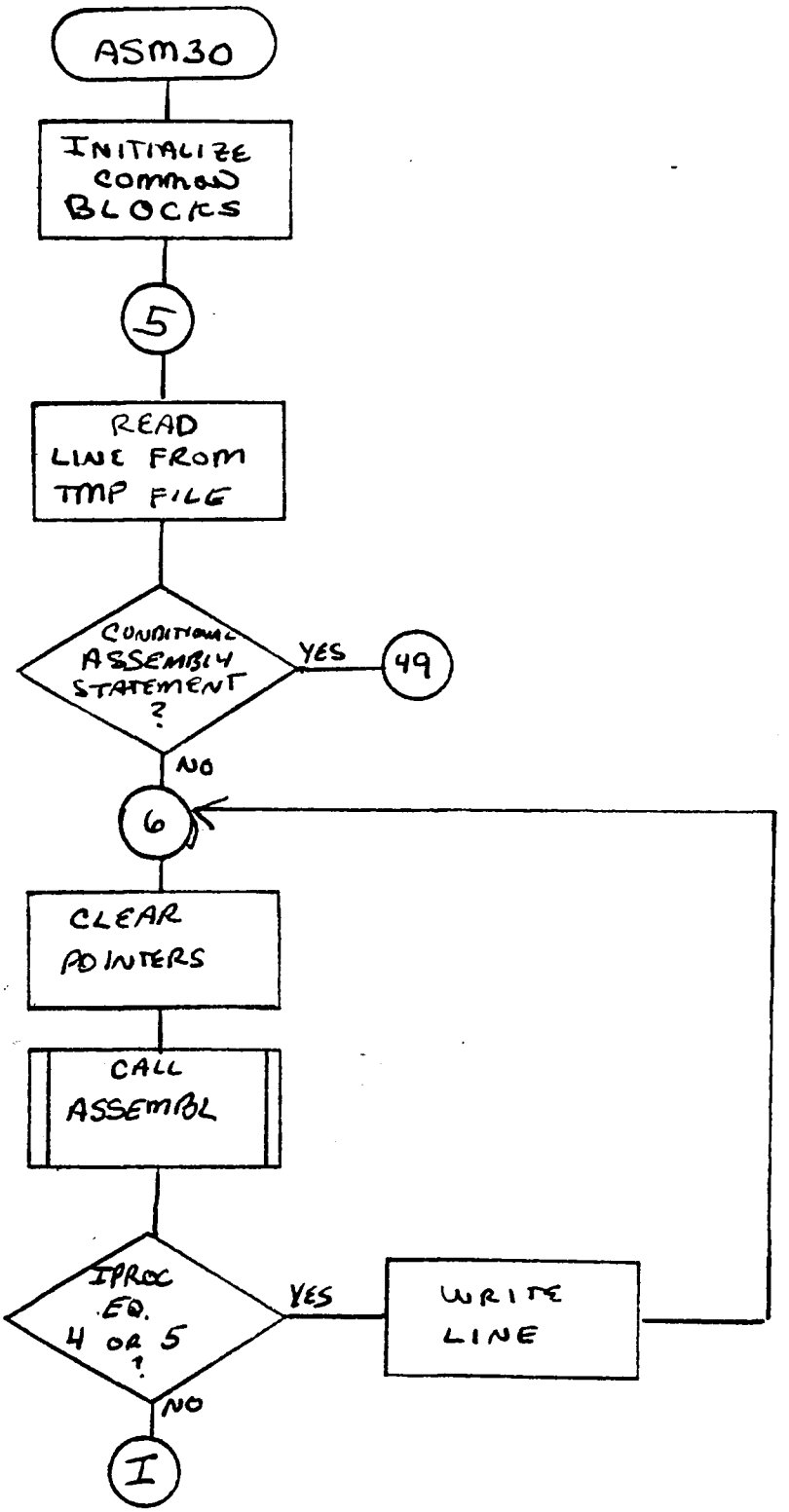
This Fortran routine writes information to the temporary file, TMP, for use during pass two. In addition to the source line it writes out the NOSAVE indicator, the macro call level (MCLC), and the macro error number (MACERR). This function is performed as a subroutine not because it is a complex procedure, but rather to ensure a uniform format on the temporary file and to facilitate changes to this format.

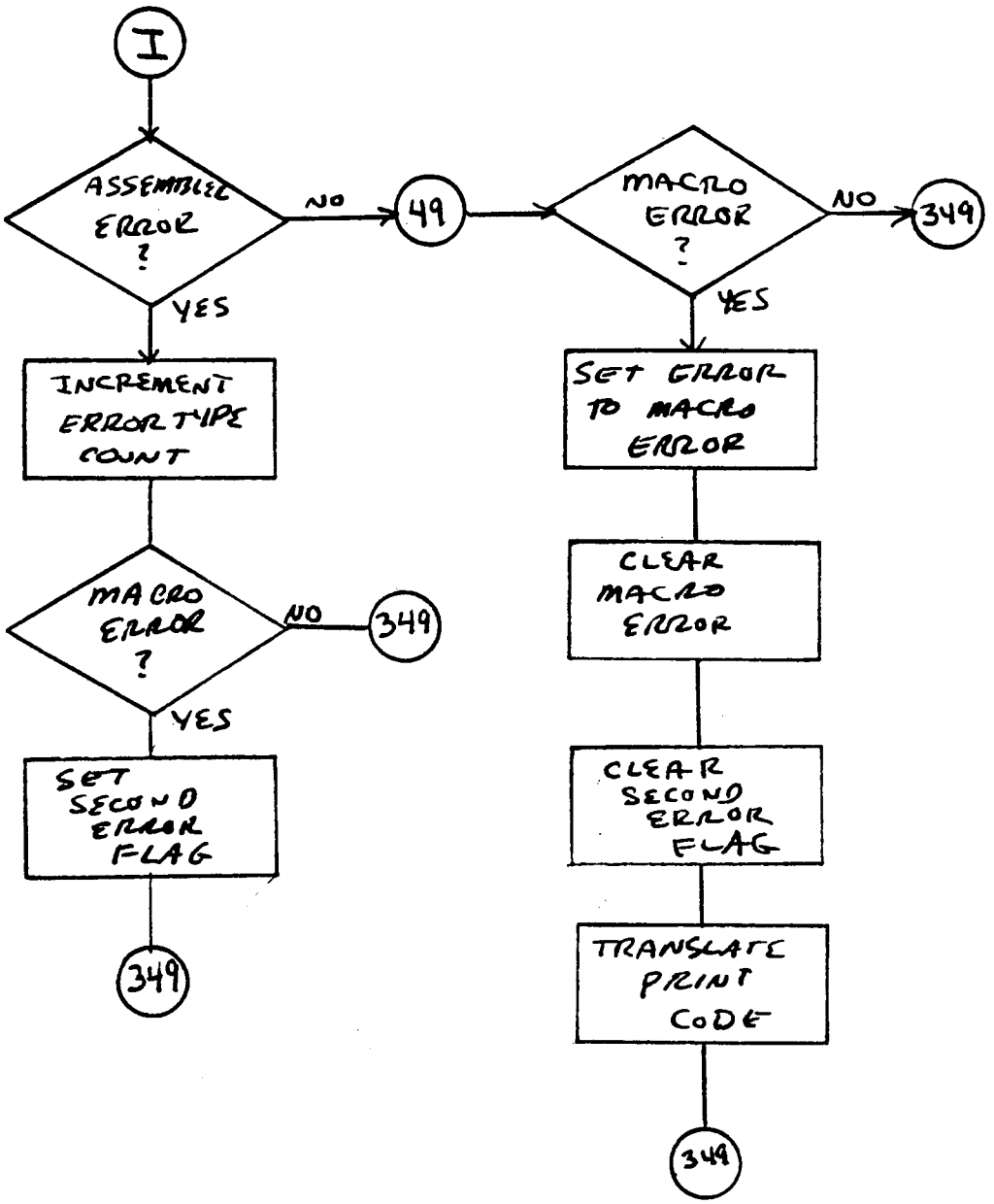


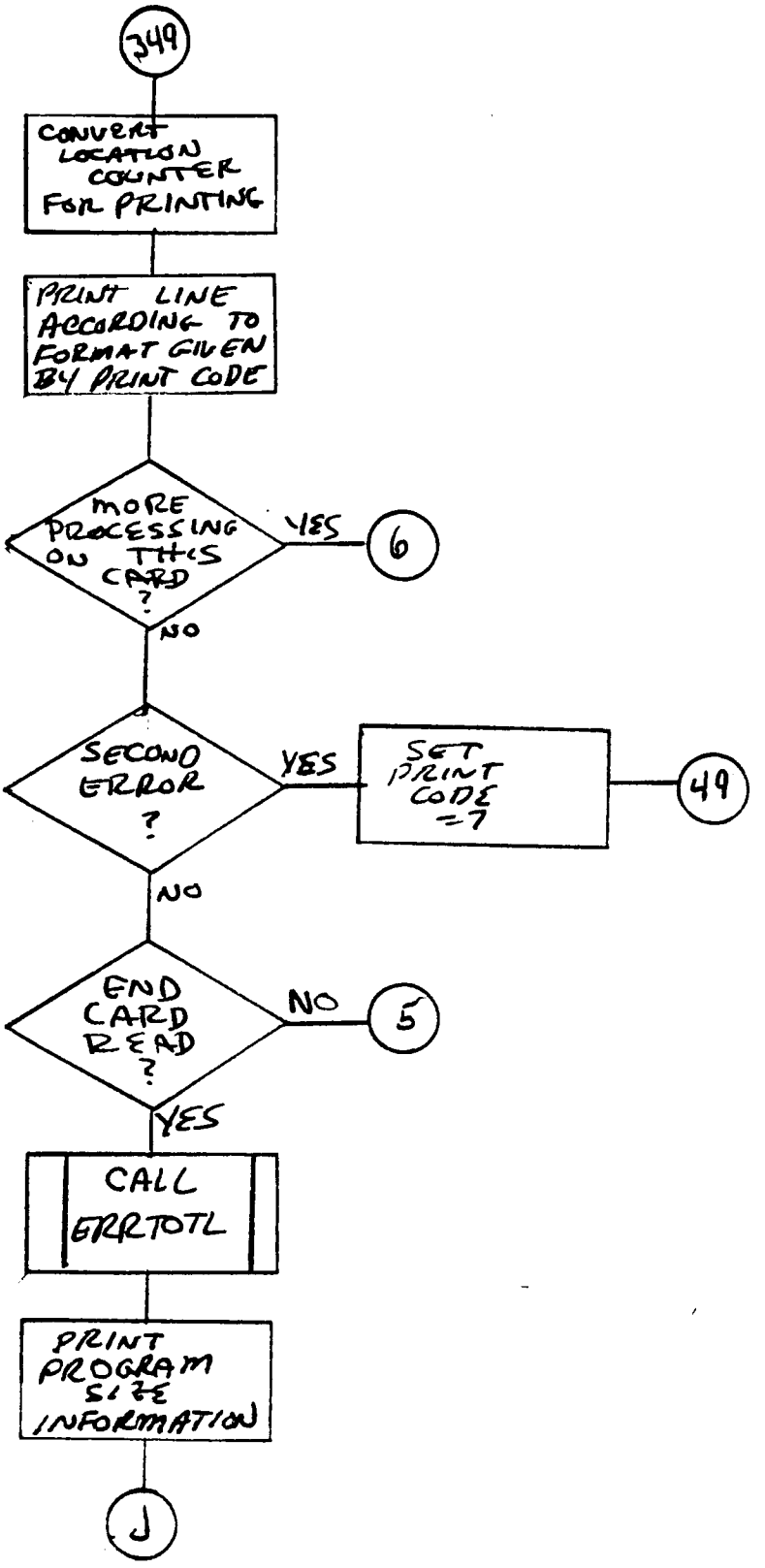
3.4.22 Program ASM30

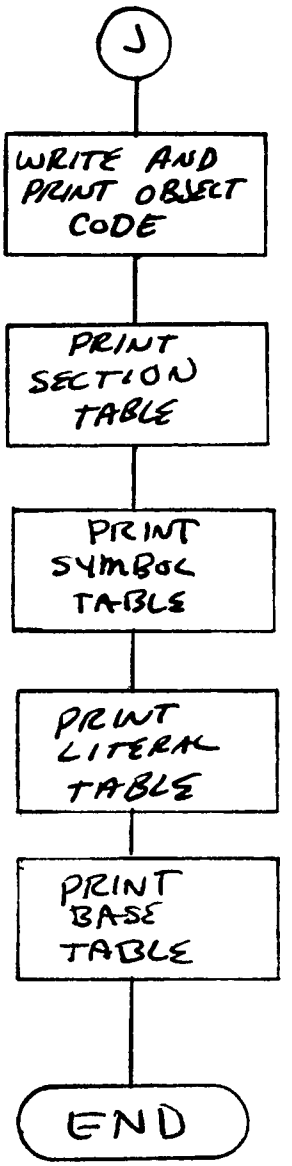
This Fortran program is the main routine of pass two of LUIAS. It reads lines from the TMP file and passes them to ASSEMBL if more processing is required. ASSEMBL sets a flag if the card needs additional processing by ASSEMBL after printing. ASM30 prints lines according to a format specified by ASSEMBL. An error number is printed if an error is detected by ASSEMBL or if a macro error occurred during pass one.

When an END card is encountered, ERRTOTL is called to print the definitions of each error encountered in the program. Then the object code generated is printed as it is dumped to the PGM file. Following the object code are listings of the section, symbol, literal, and base register tables. After the tables are printed control is returned to the ROOT phase.


ASM30
INITIALIZE COMMON BLOCKS
5
READ LINE FROM TMP FILE
CONDITIONAL ASSEMBLY STATEMENT ?
YES
49
NO
6
CLEAR POINTERS
CALL ASSEMBL
IPROC .EQ. 4 OR 5 ?
YES
WRITE LINE
NO
I

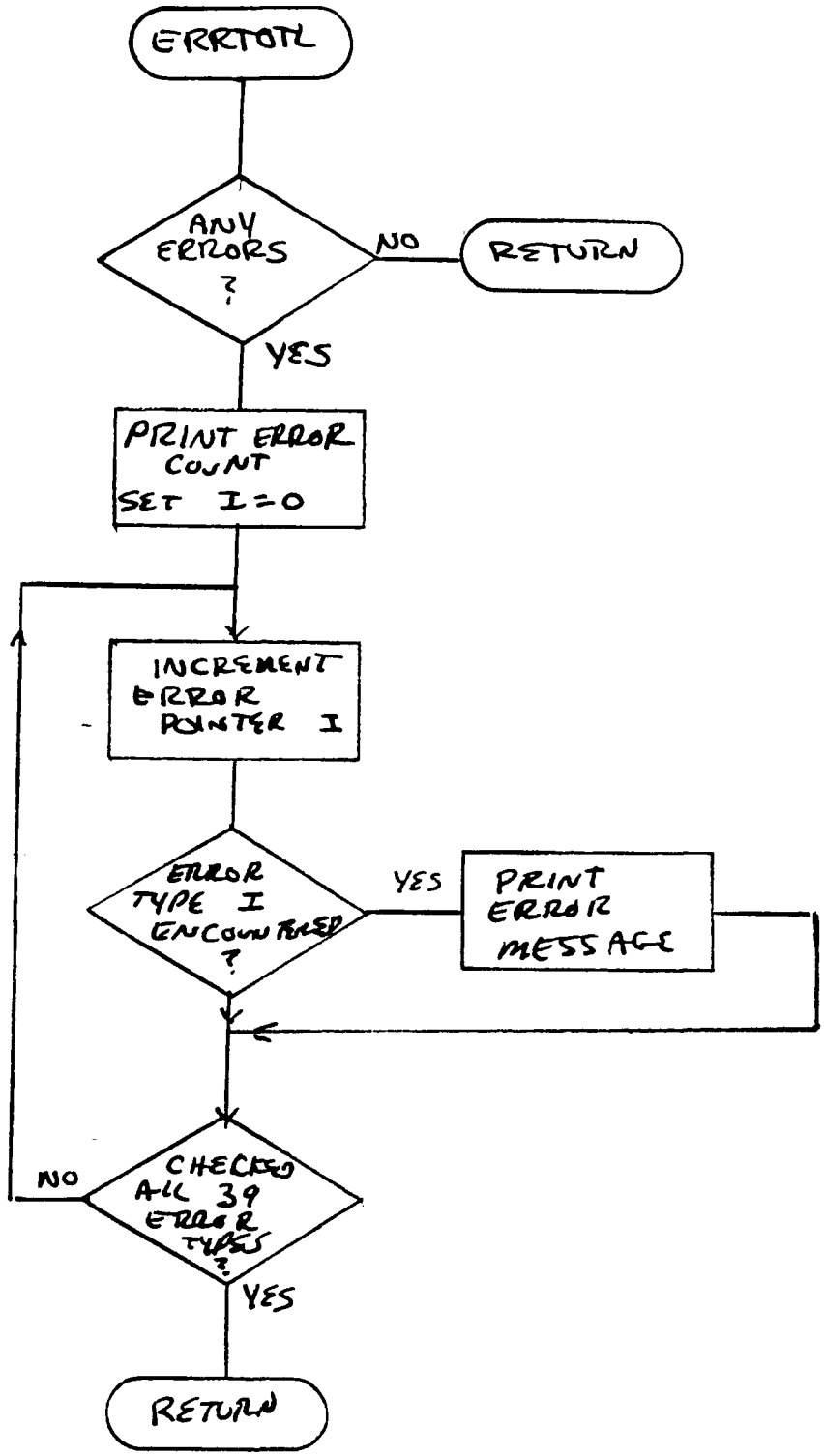

I
ASSEMBLER ERROR ?
NO
49
MACRO ERROR ?
NO
349
YES
YES
INCREMENT ERROR TYPE COUNT
SET ERROR TO MACRO ERROR
MACRO ERROR ?
NO
349
YES
CLEAR MACRO ERROR
SET SECOND ERROR FLAG
CLEAR SECOND ERROR FLAG
TRANSLATE PRINT CODE
349
349


349
CONVERT LOCATION COUNTER FOR PRINTING
PRINT LINE ACCORDING TO FORMAT GIVEN BY PRINT CODE
MORE PROCESSING ON THIS CARD ?
YES
6
NO
SECOND ERROR ?
YES
SET PRINT CODE =7
49
NO
END CARD READ ?
NO
5
YES
CALL ERRTOTL
PRINT PROGRAM SIZE INFORMATION
J


J
WRITE AND PRINT OBJECT CODE
PRINT SECTION TABLE
PRINT SYMBOL TABLE
PRINT LITERAL TABLE
PRINT BASE TABLE
END

3.5.23 Subroutine ERRTOTL

This Fortran routine prints the meaning of each error which occurred in the program. If no errors occurred control is immediately returned to the calling program. In addition to printing the error definitions, it prints the total number of errors encountered in the program.

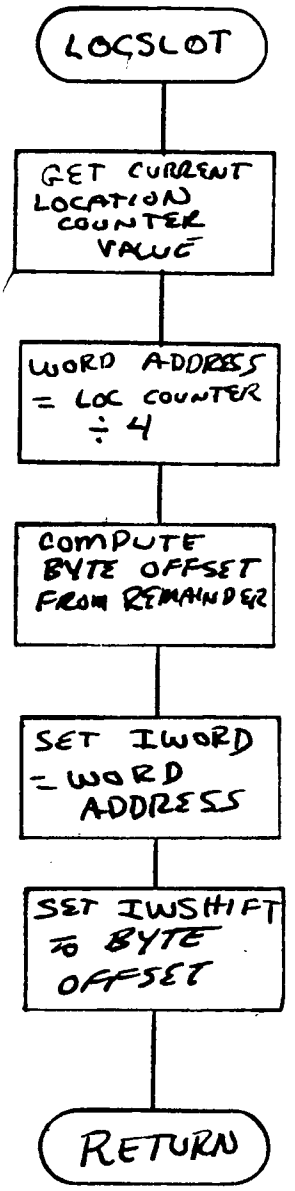

ERRTOTL
ANY ERRORS ?
NO
RETURN
YES
PRINT ERROR COUNT
SET I=0
INCREMENT ERROR POINTER I
ERROR TYPE I ENCOUNTERED ?
YES
PRINT ERROR MESSAGE
CHECKED ALL 39 ERROR TYPES ?
NO
YES
RETURN

3.4.25 Subroutine LOCSLOT

This COMPASS routine converts the current location counter value into pointers for use by the SLOT subroutine of ASSEMBL. The values returned are an index into the array ICONT and a pointer to the location in the word in the array. This routine performs its function entirely independently of the SLOT routine. This routine is called by the CSECT and ORG processing portions of ASSEMBL.

3.4.25 Subroutine LOCSLOT

This COMPASS routine converts the current location counter value into pointers for use by the SLOT subroutine of ASSEMBL. The values returned are an index into the array ICONT and a pointer to the location in the word in the array. This routine performs its function entirely independently of the SLOT routine. This routine is called by the CSECT and ORG processing portions of ASSEMBL.


LOCSLOT
GET CURRENT LOCATION COUNTER VALUE
WORD ADDRESS = LOC COUNTER ÷ 4
COMPUTE BYTE OFFSET FROM REMAINDER
SET IWORD = WORD ADDRESS
SET IWSHIFT TO BYTE OFFSET
RETURN

3.4.26 Changes to ASSEMBL

Due to the size and complexity of the COMPASS subroutine ASSEMBL, only the changes that were made to it will be described here. Modifications were made to call macro subroutines, in addition to ones to evaluate CNOP, CSECT and DSECT statements. A number of minor changes were made to improve the performance of various statements.

When a CSECT or DSECT statement is encountered the current value of the location counter is stored in the CSECT table in the entry reserved for the current CSECT. The table is then searched to determine if the new CSECT is the start of a section or the continuation of an existing one. If it is new, the location counter is zeroed; if not the location counter value is retreived from the table. The new section name and number are stored in common and a flag is set to distinguish a CSECT from a DSECT. This flag is tested by the SLOT routine before code is written to the object code array, ICONT. If it is set to indicate a DSECT, no code is entered into the array.

Related changes were made to the symbol table and the expression evaluation routine. The section number of all relocatable labels is included in the symbol table. When an expression is evaluated, the section number is checked to ensure that a valid address can be calculated. The section is also checked when base registers are assigned; the assigned base register must cover the referenced section.

At the end of the first pass, the section table and symbol table are processed together. The starting address of each CSECT is calculated from the lengths of lower numbered CSECTs. The entries in the symbol table are then relocated by adding the starting address of each CSECT to all labels defined in that CSECT. DSECTs require no processing at this time because all DSECTs start at address zero.

During the second pass, CSECT and DSECT statements merely cause the location counter value to change appropriately. No further processing is required on the symbol table.

When the CNOP instruction is encountered, its operand is loaded into a register. A table specifying boundary alignments is searched until a matching operand is found. If one of the six valid operands is not specified, an error flag is set and processing continues with the next source statement. The boundary alignment is given as a two or three bit number, depending on whether a full word or double word boundary is requested. These bits are compared against the least significant bits of the location counter. If there is a match, processing ceases; if not, a two-byte NOPR (0700) is generated, the location counter advanced, and the comparison tried again. This procedure is guaranteed of producing the required boundary after a maximum of six bytes have been generated.

When an A-type constant is encountered in a DC statement, routine AVAL is called. This routine backs up the card column

pointer to the start of the constant operand. AVAL then calls the expression evaluation routine which returns an address value for the operand or an error flag if the operand could not be evaluated. The address is returned by AVAL to the DC processing code.

If a zero duplication factor is found by the DC/DS handler, a special routine is called. This routine finds the data type in a table which specifies the address of one of four routines. These routines perform the operations needed to advance the location counter to the proper boundary and to assign a length to the label on the card. (If a double word boundary is specified, the length associated with the label will only be four instead of eight bytes because an eight byte length can not be stored in the symbol table.)

To process the macro pseudo-ops, additional entries were placed in the pseudo-op table. The routines associated with these entries do minimal processing. In most cases the index used by WORDS is set, and then a call is made to an external subroutine which performs the bulk of the processing. During the second pass, the recognition of these pseudo-ops merely sets the print code and an error code; valid macro pseudo-ops are not passed to ASSEMBL during the second pass.

One of the most extensive changes to ASSEMBL did not produce any new code, but actually removed some. This was the conversion of the routine to two separate routines for

pass one and pass two. Instead of having two source decks with much similar code, conditional assembly statements were inserted in the ASSEMBL source deck. These statements indicate to the COMPASS assembler which of the lines are to be included in each pass. Two assemblies are now required to assemble the entire routine; one generates pass one, and the other, pass two.

3.4.27 Assembler Files

LUIAS uses six files, three of which are avilable to the user. Eighty character card images are read from the file INPUT; printed output is written to OUTPUT. Assembled object code is written to the PGM file. These files may be specified by the user on the LUIAS call card. Alternate file names may be entered as positional parameters in the order mentioned above. The default file names are given as follows:

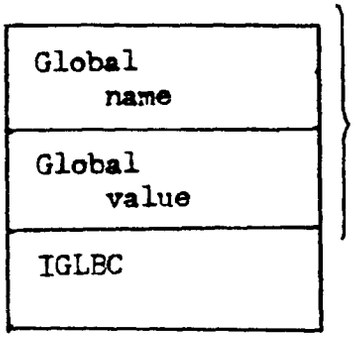
LUIAS(INPUT,OUTPUT,PGM)

Three other files are for internal use by the assembler. TMP holds information, including the source line, between the two passes of the assembler. MDT is a random access file used for storing macro definition cards; BASE is used for holding base register information that is to be used for generating a base table.

3.4.28 Common Blocks

The following pages give the formats of the various common blocks used by LUIAS. With each entry is a brief description of its use and the name by which it is most frequently addressed. Common blocks are used throughout LUIAS to reduce parameter passing between subroutines and to facilitate modularity.

Common Block /GLOBALS/

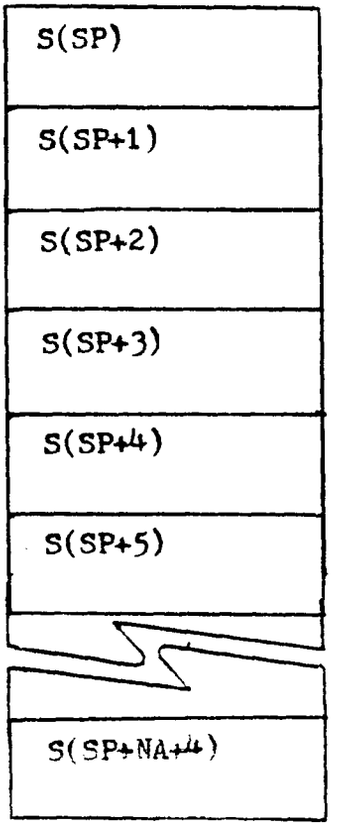


IGBLVAL

repeated 50 times

pointer to next free entry in IGBLVAL

STACK Frame format



backward link to last stack frame

MDTI(N) - start of current macro definition in MDT file

current ACTR value for this call

$SYSNDX value for this call

$SYSECT value for this call

first formal parameter value

last formal parameter value

Common Block /SUBSYM/

ISYMVAR	name of set symbol to be replaced
JSTART	current location in array C2
LIM	length of current symbol

Common Block /CARD/

C1 80 words	Input line in 80A1 format
C2 120 words	used for expansion of input line

Common Block /ICONT/

ICONT 2500 words	array for object code produced by assembler

Blank Common

MNT 100 words	Macro Name Table
IXM 1001 words	Index for mass storage file
ALA1 1000 words	Argument List Array
S 1000 words	Stack
ALA1I 100 words	Argument List Array Index
MDTI 100 words	Macro Definition Table Index

Common Block /CONDASM/

IWHERE	sequence symbol specified by AGO or AIF
IFIND	sequence symbol from current card
NOSAVE	=0 if card to be saved for pass 2 =2 if next card is not to be read

Common Block /DELIM/

IDOL	1H$
IPER	1H.
IEQU	1H=
IPLUS	1H+
IMIN	1H-
IAST	1H*
ISLASH	1H/
ICOMMA	1H,
ILPAR	1H(
IRPAR	1H)
ISPACE	1H

Common Block /LITINFO/

ILIT 150 words 3 words per entry	
ABCDEF G	The literal table
H I J K L	
M N	
ILITPNT	pointer to entry in literal table to be printed
ILEND	=1 if end of table reached
ILPRNT1	current literal in display code
ILPRNT2	current literal's core address
ILPRNT3	contents of literal core
ILPRNT4	address in binary
IKNOW	=0 if contents of literal core address are not known
ILLEN	length of literal in bytes

Bit Usage for Literal Table Entries

A =1 if entry is in use	(1 bit)
B =1 if entry in previous literal block	(1 bit)
C =1 if end of this literal block	(1 bit)
D =1 if entry has been printed	(1 bit)
E =1 if core contents are known	(1 bit)
F unused	(1 bit)
G literal in display code	(54 bits)
H unused	(30 bits)
I section number of literal block	(4 bits)
J literal boundary 0,1,2,3 = byte, halfword, fullword, double word	(2 bits)
K length of literal	(4 bits)
L core address	(20 bits)
M unused	(12 bits)
N core contents of literal location	(48 bits)

Common Block /MACINFO/

SP	stack pointer to current frame
WORD	last word found by WORDS
ALA1C	next free entry in ALA1I
MNTC	next free entry in MNT
MDTC	next free entry in MDT file
MDLC	macro definition level counter
N	index for current macro call in MNT array
NA	number of actual arguments
INDEX	current position in card used by subroutine WORDS
LABEL	sequence label found by WORDS
LARG	length of WORD found by WORDS
PLUSFL	TRUE if continuation card
NODUM	unused
MCLC	macro call level counter
MNTCALL	=1 if ASSEMBL finds an undefined operation code
MACNOW	=1 if macro definition
ISWORD	start of WORD found by WORDS

Common Block /BASINFO/

IBTAB 6 words		
A	B	C
IBADD		
IBNUM		
IBCONT		

the base table

=1 if a base register assignment has been made

number of last base register assigned

contents of base register just assigned

A =1 if base register is available (1 bit)
B section number that this register covers (4 bits)
C contents of base register (55 bits)

Common Block .SECTINF/

CSECTAB 16 words	
A	B
SYSECT	
CURSECT	
SECTGEN	

the section table

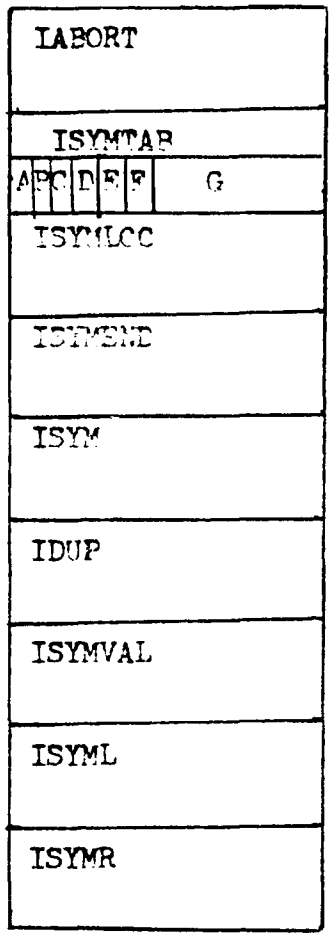
name of the current section (for $SYSECT)

current section number (0-15)

=0 if CSECT =1 if DSECT
also used for section length during printing

A location counter for this section (30 bits)
B Name for this section
right justified, zero filled (30 bits)

Common Block /SYMINFO/



≠0 if abort condition occurs during pass 1; =0 normally

Symbol table
100 words

Current pointer into ISYMTAB

=1 if end of table reached

current symbol for printing

=1 if current symbol duplicated

current symbol value
(right justified)

symbol length (1-6)

=1 if relocatable
=0 if absolute

A	=1 if entry is in use	(1 bit)
B	=1 if entry is duplicated	(1 bit)
C	=1 if entry is absolute	(1 bit)
D	section number (0-15)	(4 bits)
E	symbol length in bytes	(3 bits)
F	symbol value in binary	(20 bits)
G	symbol in display code	(30 bits)

Common Block /INFO/

ICARD 8 words	Storage of current line in 8A10 format
IERR	if no error IERR=0 if error IERR= 1-17
ILOC	current location counter value
IOLOC	location counter value at start of current line
ICODE 2 words	Object code for this instruction
IEND	=1 if END card found
IPROC	=0 if card is finished ≠0 if more processing required
IPRINT	Code indicating format to use - to print this line
IPASS	=0 if pass 1 =1 if pass 2
IDEL	unused
ILABL	=1 if there is a label on the current card
ICHAR	pointer to character in current line =-1 if end of card reached
IERSEV	unused
IWORD	Index to ICONT array for current word
IOUT	Index to ICODE for current word
IWSHIFT	Index to bit position of current word of ICONT (54-12)
IOSHIFT	Index to bit position of current word of ICODE (54-0)
ISTART	Base address of assembly

3.5 Suggestions for Improving LUIAS

A number of changes could be made to LUIAS to improve its efficiency and usefulness. This section outlines a number of improvements which could have a major effect upon the operation of LUIAS.

Because LUIAS is the result of the programming efforts of three different people working independently over a span of 1½ years, there is some duplication of code and procedures. Removing extra code and improving the interfacing of some routines would improve execution time and reduce memory size. Other changes could increase the power of the assembler and produce an assembler more compatible with the IBM assembler.

3.5.1 Efficiency and Memory Size

At the present time subroutine WORDS is called whenever a label, opcode, or part of an operand is required by the macro processor. Calls to this routine could be reduced if subroutines WORDS or ASUBS stored the label and opcode in separate words in a common block when a line is first read. Calls to WORDS would then return words found in the operand only. Comments could then be permitted on most lines processed by WORDS if WORDS returned an END OF CARD condition when a blank is found in the operand. The SETB pseudo-op would have to be

an exception to this rule, because blanks are permitted as delimiters in a Boolean expression.)

Subroutine GETCHAR could be modified to take advantage of the array C1 which contains the current line as single characters during the first pass. The array is not available during the second pass so the old GETCHAR routine would have to be retained there.

Table searches throughout the assembler could be improved. All are now performed as sequential searches, and could certainly be improved with a more efficient searching algorithm. In particular, the machine operation table (MOT), pseudo-operation table (POT), macro name table (MNT), and the symbol table are searched frequently. Every source line forces a search of the MOT and POT, the two tables which would be the easiest to optimize. These two tables are static and thus could be arranged in an optimal sequence for the searching algorithm chosen.

A number of macro tables occupy a lot of memory and are also searched frequently during macro calls and definitions. These could be combined into a unified table containing a block of entries for each definition. Among these arrays are the Argument List Array (ALA1), Macro Name Table (MNT), Argument List Array Index (ALA1I), and Macro Definition Table Index (MDTI). Combining these into a single array could do away with the need for indexes to several different arrays. The combined array and possibly the stack could be written to a random access file to be read into memory only as needed. This

change would increase I/O time but decrease table searching time and memory size.

The random access file handling method could be improved to reduce memory requirements. The present structure requires a fixed array containing one entry for each record which may be written to the file. If subindexes were used the size of the array could be reduced considerably.

The COMPASS routine ASSEMBL, which does the bulk of the processing for both passes, contains some extraneous code. In particular there is code in it which is needed during one pass only but is included in both passes. There is probably a good bit of processing done during either pass which needs not be done during that pass. Most of this kind of code resulted because the assembler was originally written as a single program with no overlays, with no concern given to the size or execution time of the two passes. When the assembler was converted to a two-pass, two-overlay structure, not all the code was properly identified as belonging to only one pass. (A flowcharting program for COMPASS source would be a tremendous aid in the identification of code by pass.)

It is possible to reduce the duplication of processing in the two passes by writing more information to the temporary file along with the source line. For instance, saving the op-code type and the operand starting column would eliminate the need for searching the opcode tables in both passes.

The efficiency of the Fortran code could be improved in a combination of two ways. Routines which are used often (ASUBS and WORDS, to name two) could be recoded in COMPASS and other routines could be optimized. Conversion to the optimizing FTN compiler would be straightforward. The most significant problem is to ensure that COMPASS subroutines with parameters are changed accordingly and to ensure that no COMPASS routine destroys register A0. The only COMPASS routines with parameters are CONVERT, UNBLNK, and SECTPRN. A minor change involving EOF testing would have to be made to all Fortran routines which read from the input file.

Once LUIAS is converted to the FTN compiler, it could be compiled with the optimization option specified. Hopefully, this would produce faster executing object code.

3.5.2 Listing Control

Currently there are no options available with regard to the listings produced by the assembler. On every run it produces the source listing, the object code listing, and tables of sections, symbols, literals and base registers.

Some kind of listing options would improve the readibility of the printed output generated by LUIAS. (They would also decrease the number of pages printed and thus save money and trees.) IBM has a pseudo-op which allows the user to selectively suppress all listings or just the expansion of macro calls.

In addition to these two options it would be useful to select which of the tables and object code listings are to be printed.

3.5.3 DC/DS Statements and Literals

Another area in which LUIAS limitations hinder the user is the evaluation of DC statements and literals. Closely involved with the routines to evaluate these statements is the expression evaluation routine. Altogether these routines take nearly 1000 source lines of COMPASS code, accounting for nearly one-third of the code in the main assembler routine ASSEMBL. Due to the number of limitations of these routines, it would probably be best to entirely rewrite this code in order to improve it.

The most significant problem is that of the short maximum length permitted in DC operands and literals. The six character limit is a direct result of attempting tc store the entire operand in a single 60-bit word. While this simplifies literal processing, it serves no useful purpose in DC statement evaluation.

Recoding the expression routine would allow the recognition of other than decimal constants, in addition to properly interfacing with the new literal handling code.

New literal routines would process any length constants, sharing as much code as possible with the DC statement routines.

The format of the literal table should be changed so that variable length literals could be stored. Also the section in which a literal is referenced should be stored so that it can be generated in the proper CSECT when an LTORG is encountered.

3.5.4 Execution Cost

Another problem which should be investigatied is that of total cost of an assembly. At present the cost seems high. It would be appropriate to consult the Computing Center to determine which areas are causing the high cost (PP, CP, IO time, etc.) and then to concentrate efforts on improving that facet of LUIAS.

Bibliography

1. "COMPASS Version 3 Reference Manual," Control Data Corporation, 1974

2. Horey, Leonard I., "The Lehigh University IBM/360 Simulator," Master's Thesis, Lehigh University, 1975.

3. "IBM System/360 Operating System Assembler Lnaguage," Form C28-6514-5. International Business Machines Corporation, San Jose, California. 1964

4. Maroudas, George S., "Digital Logic Simulation and Macro Processing," Master's Thesis, Lehigh University, 1975.

5. Seager, Henry P., "The Lehigh University IBM 360 Assembler," Master's Thesis, Lehigh University, 1975.

Hex	Disp	CS	TTY	Hex	Disp	CS	TTY	Hex	Disp	CS	TTY	Hex	Disp	CS	TTY
40	55	b	b	50	67	^	&	60	46	-	-	F0	33	0	0
C1	01	A	A	D1	12	J	J	61	50	/	/	F1	34	1	1
C2	02	B	B	D2	13	K	K	F2	23	S	S	F2	35	2	2
C3	03	C	C	D3	14	L	L	E3	24	T	T	F3	36	3	3
C4	04	D	D	D4	15	M	M	E4	25	U	U	F4	37	4	4
C5	05	E	E	D5	16	N	N	E5	26	V	V	F5	40	5	5
C6	06	F	F	D6	17	O	O	E6	27	W	W	F6	41	6	6
C7	07	G	G	D7	20	P	P	E7	30	X	X	F7	42	7	7
C8	10	H	H	D8	21	Q	Q	E8	31	Y	Y	F8	43	8	8
C9	11	I	I	D9	22	R	R	E9	32	Z	Z	F9	44	9	9
4A	72	<	<	5A	66	V	!	6A	75	≥	\	7A	00	b	b
4B	57	.	.	5B	53	$	$	6B	56	,	,	7B	60	≡	#
4C	74	≤	@	5C	47	*	*	6C	63	:	:	7C	70	↑	•
4D	51	(	(	5D	52	)	)	6D	65	→	←	7D	61	⊏	⊏
4E	45	+	+	5E	77	;	;	6E	73	>	>	7E	54	=	=
4F	62	]	]	5F	76	¬	↑	6F	71	↓	?	7F	64	≠	"

Table of character codes showing hexadecimal and CDC display code values and external characters as they appear on the central site printer and on a teletype.

Appendix B. Mnemonics Assembled by LUIAS

RR Instructions

SPM BALR BCTR BCR LPR LNR CLR OR XR LR CR AR

SR MR DR ALR SLR

SI Instructions

SIO TS TM MVI NI CLI OI XI

RX Instructions

STH LA STC IC EX BAL BC LH AH SH MH CVD SL

CVB ST N CL O X C A S M D AL

RS Instructions

BXH BXLE LM STM SRL SLL SRA SLA SRDL SRDA SLDA

SS1 Instructions

MVN MVC MVZ NC CLC OC XC TR TRT

SS2 Instructions

MVO PACK UNPK

Extended Mnemonics

B BR NOP NOPR BH BL BE BNH BNL BNE BO BP

BM BZ BNP BNM BNZ BNO

Pseudo Instructions

START ORG USING DROP LTORG END CSECT DSECT CNOP EQU

DS DC IO LCLA LCLB GBLA GBLB SETA SETB AGO

AIF ANOP ACTR MACRO MEND

Appendix C Compilation Procedure and Load Map for LUIAS

LUIAS contains three overlays which must be compiled in a specific sequence. Certain routines must go in specific overlays and overlays must be loaded in sequence. Loader OVERLAY cards must be the first card of each overlay compiled. These cards inform the loader of the boundaries of each overlay. The main routine of each overlay must be a Fortran PROGRAM rather than a SUBROUTINE, in order for each overlay to specify a transfer address. Two versions of ASSEMBL may be assembled. For the first pass version the following card must be inserted after the third card in the source deck:

```
PASS1    SET    1
```

For the second pass:

```
PASS1    SET    -1
```

It is easiest to insert these cards if the source to ASSEMBL is stored on a permanent file in UPDATE format.

The remainder of this section gives a listing of the subroutines needed in each overlay. A load map is included showing how LUIAS is loaded and the system routines which are loaded in each overlay.

```
ROOT PHASE     -     OVERLAY(ASMOVER,0,0)
        ASSMB        CONVERT
INITIALIZATION PHASE   -   OVERLAY(ASMOVER,1,0)
        INIT10
PASS ONE     -     OVERLAY(ASMOVER,2,0)
        ASM20        WRITE        MACMNT       AMACRO       DUMMY
        ARGUMNT      READ         ASUBS        REPSYM       WORDS
        SET          ARITH        AIFEVAL      FACTR        GBL
        AGOPROC      LABFND       NUMBCHK      AERROR       ABORT
        UNBLNK       ASSEMBL(pass 1)
PASS TWO     -     OVERLAY(ASMOVER,3,0)
        ASM30        ERRTOTL      LITPRIN      GETSYM       SECTPRN
        LOCSLOT      ASSEMBL(pass 2)
```

LOAD MAP - ASSMB CYBER LOADER 1.0-R406 11/24/75 09.15.59. PAGE 1
OVERLAY(ASMOVER,0,0)

-------- OVERLAY(ASMOVER,0,0)

FWA OF THE LOAD 101
LWA+1 OF THE LOAD 20434

TRANSFER ADDRESS -- ASSMB 664

PROGRAM AND BLOCK ASSIGNMENTS.

BLOCK	ADDRESS	LENGTH	FILE	DATE	PROCSSR	VER	LEVEL	HARDWARE	COMMENTS
/INFO/	101	32							
/ERRINF/	133	50							
/SYMINFO/	203	154							
/LITINFO/	357	236							
/BASINFO/	615	23							
/SECTINF/	640	23							
ASSMB	663	11302	A1	11/24/75	RUN	2	.3	75073	
CONVERT	12165	14	A1	11/24/75	COMPASS	3.		75161	
ACCDER	12201	12	SL-RUN2P3	03/14/75	COMPASS	3.		74224	
SYS-TIM	12213	31	SL-RUN2P3	03/14/75	COMPASS	3.		74224	
OUTPTS	12244	70	SL-RUN2P3	09/11/75	COMPASS	3.		75161	
OVERLAY	12334	123	SL-RUN2P3	09/11/75	COMPASS	3.		75161	
REMARK	12457	4	SL-RUN2P3	09/11/75	COMPASS	3.		75161	
GETBA	12463	17	SL-RUN2P3	09/11/75	COMPASS	3.		75161	
IFENDF	12502	66	SL-RUN2P3	09/11/75	COMPASS	3.		75161	
INPUTC	12570	121	SL-RUN2P3	09/11/75	COMPASS	3.		75161	
KODER	12711	1261	SL-RUN2P3	09/11/75	COMPASS	3.		75161	
KRAKER	14172	1044	SL-RUN2P3	09/11/75	COMPASS	3.		75161	
OUTPTC	15236	72	SL-RUN2P3	09/11/75	COMPASS	3.		75161	
SIOB	15330	1475	SL-RUN2P3	09/11/75	COMPASS	3.		75161	
SYSTEM	17025	1063	SL-RUN2P3	09/23/75	COMPASS	3.		75161	
UCLOAD	20110	265	SL-NUCLEUS	10/16/75	COMPASS	3.		75161	L75291 LOADER USER CALL INTERFACE ROUTINE.
SYS.RM	20375	37	SL-NUCLEUS	10/20/75	COMPASS	3.		75161	PROCESS SYSTEM REQUEST.

-------- OVERLAY(ASMOVER,1,0)

FWA OF THE LOAD 20435
LWA+1 OF THE LOAD 20636

TRANSFER ADDRESS -- INIT10 20436

PROGRAM AND BLOCK ASSIGNMENTS.

BLOCK	ADDRESS	LENGTH	FILE	DATE	PROCSSR	VER	LEVEL	HARDWARE	COMMENTS
INIT10	20435	116	A2	11/06/75	RUN	2	.3	75073	

OVERLAY(ASMOVER,1,0)

BLOCK	ADDRESS	LENGTH	FILE	DATE	PROCSSR	VER	LEVEL	HARDWARE	COMMENTS
REWINM	20553	63	SL-RUN2P3	09/11/75	COMPASS	3.	75161		

OVERLAY(ASMOVER,2,0)

ENA OF THE LOAD 20435
LWA+1 OF THE LOAD 37236

TRANSFER ADDRESS -- ASM20 20500

PROGRAM AND BLOCK ASSIGNMENTS.

BLOCK	ADDRESS	LENGTH	FILE	DATE	PROCSSR	VER	LEVEL	HARDWARE	COMMENTS
/COMDASM/	20435	3							
/SUBSYM/	20440	3							
/MACINFO/	20443	21							
/DELIM/	20464	13							
ASM20	20477	245	A3	11/24/75	RUN	2	.3	75073	
WRITE	20744	32	A3	11/24/75	RUN	2	.3	75073	
/CARD/	20776	310							
MACMNT	21306	305	A3	11/24/75	RUN	2	.3	75073	
AMACRO	21613	225	A3	11/24/75	RUN	2	.3	75073	
DUMMY	22040	212	A3	11/24/75	RUN	2	.3	75073	
ARGUMNT	22252	241	A3	11/24/75	RUN	2	.3	75073	
READ	22513	106	A3	11/24/75	RUN	2	.3	75073	
ASUBS	22621	303	A3	11/24/75	RUN	2	.3	75073	
/GLOBALS/	23124	145							
REPSYM	23271	271	A3	11/24/75	RUN	2	.3	75073	
MACDS	23562	151	A3	11/24/75	RUN	2	.3	75073	
SET	23733	66	A3	11/24/75	RUN	2	.3	75073	
ARITH	24021	262	A3	11/24/75	RUN	2	.3	75073	
AIFEVAL	24303	364	A3	11/24/75	RUN	2	.3	75073	
FACTR	24667	42	A3	11/24/75	RUN	2	.3	75073	
GBL	24731	105	A3	11/24/75	RUN	2	.3	75073	
AGOPROC	25036	116	A3	11/24/75	RUN	2	.3	75073	
LABFND	25154	32	A3	11/24/75	RUN	2	.3	75073	
NUMBCHK	25206	74	A3	11/24/75	RUN	2	.3	75073	
APERCOR	25302	16	A3	11/24/75	RUN	2	.3	75073	
ABORT	25320	56	A3	11/24/75	RUN	2	.3	75073	
UNBLNK	25376	7	A3	11/24/75	COMPASS	3.	75161		
ASSEMBL	25405	2623	A3	11/24/75	COMPASS	3.	75161		
INITMS	30230	120	SL-RUN2P3	03/14/75	COMPASS	3.	74224		
ISHIFT	30350	3	SL-RUN2P3	06/03/75	COMPASS	3.	74224		
INPUTS	30353	62	SL-RUN2P3	09/11/75	COMPASS	3.	75161		
READMS	30435	133	SL-RUN2P3	09/11/75	COMPASS	3.	75161		
WRITMS	30570	101	SL-RUN2P3	09/11/75	COMPASS	3.	75161		
//	30671	6345							

OVERLAY(ASMOVER,3,0)

OVERLAY(ASMOVER,3,0)

FWA OF THE LOAD 20435
LWA+1 OF THE LOAD 33155

TRANSFER ADDRESS -- ASM30 25342

PROGRAM AND BLOCK ASSIGNMENTS.

BLOCK	ADDRESS	LENGTH	FILE	DATE	PROCSSR	VER	LEVEL	HARDWARE	COMMENTS
/ICONT/	20435	4704							
ASM30	25341	1315	A4	11/24/75	RUN 2	.3	75073		
ERRTOTL	26656	442	A4	11/24/75	RUN 2	.3	75073		
LITPRIN	27320	65	A4	11/24/75	COMPASS	3.	75161		
GETSYM	27405	34	A4	11/24/75	COMPASS	3.	75161		
SECTPRN	27441	11	A4	11/24/75	COMPASS	3.	75161		
LOCSLOT	27452	7	A4	11/24/75	COMPASS	3.	75161		
ASSEMBL	27461	3332	A4	11/24/75	COMPASS	3.	75161		
ENDFIL	33013	57	SL-RUN2P3	09/11/75	COMPASS	3.	75161		
REMINM	33072	63	SL-RUN2P3	09/11/75	COMPASS	3.	75161		

2.044 CP SECONDS 324008 CM STORAGE USED 18 TABLE MOVES

Appendix D Sample LUIAS Assemblies

Included in this appendix are three sample programs assembled by LUIAS. These programs exhibit several features of the macro assembler.

The first is the macro MOVE. It generates the necessary MVC instructions to transfer a large block of storage.whcih contains more than the 256 bytes which can be moved with the MVC instruction.

The second is the macro EXPO. It generates the required code to raise a variable to an integer power. An interesting feature of this macro is that it is self-recursive. Each call generates a single multiplication and then calls itself. When enough multiplications have been generated to compute the desired power, the innermost macro terminates and returns control to the calling macro. The example shows that raising a variable to the fifth power will result in five levels of recursion.

The third program is an example of macros defined within macros. The macro DEFN is called to define macros SIN, COS, and TIME. The call to TIME has no parameters specified, so a warning error (33) is indicated. The macro TAN calls macros SIN and COS. This macro uses the system index ($SYSNDX) to generate unique labels within the macro.

```
CARD NO.   ERROR   LOCATION OBJECT CODE   ---------- SOURCE STATEMENTS ----------

     1                                      MACRO
     2                                      MOVE  &OP1,&OP2,&LEN
     3                                      LCLA  &MVCL,&LEFT,&COUNT,&ZERO
     4                                &COUNT SETA  0
     5                                &MVCL  SETA  256
     6                                &ZERO  SETA  0
     7                                &LEFT  SETA  (&LEN)
     8                                .LOOP  AIF   (&LEFT LE &MVCL).FIN
     9                                      MVC   &OP1+&COUNT(&MVCL),&OP2+&COUNT
    10                                &LEFT  SETA  (&LEFT-&MVCL)
    11                                &COUNT SETA  (&COUNT+&MVCL)
    12                                      AGO   .LOOP
    13                                .FIN   AIF   (&LEFT EQ &ZERO).END
    14                                      MVC   &OP1+&COUNT(&LEFT),&OP2+&COUNT
    15                                .END   ANOP
    16                                      MEND
    17              00000             MOVES  START
    18                                      USING *,15
    19                                      MOVE  BLOK1,BLOK2,100
    20  1                                   LCLA  &MVCL,&LEFT,&COUNT,&ZERO
    21  1                             &COUNT SETA  0
    22  1                             &MVCL  SETA  256
    23  1                             &ZERO  SETA  0
    24  1                             &LEFT  SETA  (&LEN)
    25  1                                   AIF   (&LEFT LE &MVCL).FIN
    26  1                                   AIF   (&LEFT EQ &ZERO).END
    27  1              00000  D263F02CF090          MVC   BLOK1+0(100),BLOK2+0
    28  1                                   ANOP
    29                                      MOVE  CHNK1,CHNK2,1000
    30  1                                   LCLA  &MVCL,&LEFT,&COUNT,&ZERO
    31  1                             &COUNT SETA  0
    32  1                             &MVCL  SETA  256
    33  1                             &ZERO  SETA  0
    34  1                             &LEFT  SETA  (&LEN)
    35  1                                   AIF   (&LEFT LE &MVCL).FIN
    36  1              00006  D2FFF0F4F8C4          MVC   CHNK1+0(256),CHNK2+0
    37  1                             &LEFT  SETA  (&LEFT-&MVCL)
    38  1                             &COUNT SETA  (&COUNT+&MVCL)
    39  1                                   AGO   .LOOP
    40  1                                   AIF   (&LEFT LE &MVCL).FIN
    41  1              0000C  D2FFF1F4F9C4          MVC   CHNK1+256(256),CHNK2+256
    42  1                             &LEFT  SETA  (&LEFT-&MVCL)
    43  1                             &COUNT SETA  (&COUNT+&MVCL)
    44  1                                   AGO   .LOOP
    45  1                                   AIF   (&LEFT LE &MVCL).FIN
    46  1              00012  D2FFF2F4FAC4          MVC   CHNK1+512(256),CHNK2+512
    47  1                             &LEFT  SETA  (&LEFT-&MVCL)
    48  1                             &COUNT SETA  (&COUNT+&MVCL)
    49  1                                   AGO   .LOOP
    50  1                                   AIF   (&LEFT LE &MVCL).FIN
```

```
51  1                                                   AIF   ($LEFT EQ $ZERO).END
52  1           00018   D2E7F3F4F8C4                    MVC   CHNK1+768(232),CHNK2+768
53  1                                                   ANOP
54                                                      MOVE  BLOK2,CHNK2,512
55  1                                                   LCLA  $MVCL,$LEFT,$COUNT,$ZERO
56  1                                   $COUNT          SETA  0
57  1                                   $MVCL           SETA  256
58  1                                   $ZERO           SETA  0
59  1                                   $LEFT           SETA  ($LEN)
60  1                                                   AIF   ($LEFT LE $MVCL).FIN
61  1           0001E   D2FFF090F8C4                    MVC   BLOK2+0(256),CHNK2+0
62  1                                   $LEFT           SETA  ($LEFT-$MVCL)
63  1                                   $COUNT          SETA  ($COUNT+$MVCL)
64  1                                                   AGO   .LOOP
65  1                                                   AIF   ($LEFT LE $MVCL).FIN
66  1                                                   AIF   ($LEFT EQ $ZERO).END
67  1           00024   D2FFF190F9C4                    MVC   BLOK2+256(256),CHNK2+256
68  1                                                   ANOP
69              0002A   07FE                            BR    14
70              0002C                   BLOK1           DS    1000
71              00090                   BLOK2           DS    1000
72              00JF4                   CHNK1           DS    2000
73              008C4                   CHNK2           DS    2000
74                                                      END
```

```
CARD NO.   ERROR   LOCATION OBJECT CODE   ---------- SOURCE STATEMENTS ----------

     1                                     MACRO                                  EXPON  2
     2                                     EXPO  &EXP                             EXPON  3
     3                                     LCLA  &N,&ONE                          EXPON  4
     4                              &ONE   SETA  1                                EXPON  5
     5                              &N     SETA  (&EXP)                           EXPON  6
     6                                     AIF   (&N LE &ONE).STOP                EXPON  7
     7                                     MR    0,2                              EXPON  8
     8                              &N     SETA  (&N-&ONE)                        EXPON  9
     9                                     EXPO  &N                               EXPON 10
    10                              .STOP  ANOP                                   EXPON 11
    11                                     MEND                                   EXPON 12
    12              00000          EXPON  START 0                                 EXPON 13
    13                                     USING *,15                             EXPON 14
    14              00000  1800            SR    0,0                              EXPON 15
    15              00002  5810F020        L     1,BASE                           EXPON 16
    16              00006  1821            LR    2,1                              EXPON 17
    17                                     EXPO  3                                EXPON 18
    18  1                                  LCLA  &N,&ONE                          EXPON  4
    19  1                           &ONE   SETA  1                                EXPON  5
    20  1                           &N     SETA  (&EXP)                           EXPON  6
    21  1                                  AIF   (&N LE &ONE).STOP                EXPON  7
    22  1           00008  1C02            MR    0,2                              EXPON  8
    23  1                           &N     SETA  (&N-&ONE)                        EXPON  9
    24  1                                  EXPO  2                                EXPON 10
    25  2                                  LCLA  &N,&ONE                          EXPON  4
    26  2                           &ONE   SETA  1                                EXPON  5
    27  2                           &N     SETA  (&EXP)                           EXPON  6
    28  2                                  AIF   (&N LE &ONE).STOP                EXPON  7
    29  2           0000A  1C02            MR    0,2                              EXPON  8
    30  2                           &N     SETA  (&N-&ONE)                        EXPON  9
    31  2                                  EXPO  1                                EXPON 10
    32  3                                  LCLA  &N,&ONE                          EXPON  4
    33  3                           &ONE   SETA  1                                EXPON  5
    34  3                           &N     SETA  (&EXP)                           EXPON  6
    35  3                                  AIF   (&N LE &ONE).STOP                EXPON  7
    36  3                                  ANOP                                   EXPON 11
    37  2                                  ANOP                                   EXPON 11
    38  1                                  ANOP                                   EXPON 11
    39              0000C  1821            LR    2,1                              EXPON 19
    40                                     EXPO  5                                EXPON 20
    41  1                                  LCLA  &N,&ONE                          EXPON  4
    42  1                           &ONE   SETA  1                                EXPON  5
    43  1                           &N     SETA  (&EXP)                           EXPON  6
    44  1                                  AIF   (&N LE &ONE).STOP                EXPON  7
    45  1           0000E  1C02            MR    0,2                              EXPON  8
    46  1                           &N     SETA  (&N-&ONE)                        EXPON  9
    47  1                                  EXPO  4                                EXPON 10
    48  2                                  LCLA  &N,&ONE                          EXPON  4
    49  2                           &ONE   SETA  1                                EXPON  5
    50  2                           &N     SETA  (&EXP)                           EXPON  6
```

```
                                                                                                                EXPON  7
                                                                                                                EXPON  8
                                                                                                                EXPON  9
                                                                                                                EXPON 10
                                                                                                                EXPON  4
                                                                                                                EXPON  5
                                                                                                                EXPON  6
                                                                                                                EXPON  7
                                                                                                                EXPON  8
                                                                                                                EXPON  9
                                                                                                                EXPON 10
                                                                                                                EXPON  4
                                                                                                                EXPON  5
                                                                                                                EXPON  6
                                                                                                                EXPON  7
                                                                                                                EXPON  8
                                                                                                                EXPON  9
                                                                                                                EXPON 10
                                                                                                                EXPON  4
                                                                                                                EXPON  5
                                                                                                                EXPON  6
                                                                                                                EXPON  7
                                                                                                                EXPON 11
                                                                                                                EXPON 11
                                                                                                                EXPON 11
                                                                                                                EXPON 11
                                                                                                                EXPON 11
                                                                                                                EXPON 21
                                                                                                                EXPON 22
                                                                                                                EXPON 23
                                                                                                                EXPON 24
                                                                                                                EXPON 25
```

```
CARD NO.   ERROR   LOCATION OBJECT CODE   ---------- SOURCE STATEMENTS ----------

     1             00800                 RECUR    START 2048
     2                                            USING *,12
     3             00800   41000C48               LA    13,SAVEAREA
     4                                   *
     5                                            MACRO
     6                                            DEFN  $NAME
     7                                   .* THIS MACRO GENERATES A MACRO WHICH DEFINES A SUBROUTINE CALL
     8                                            MACRO
     9                                   $LABEL   $NAME $PRMLST
    10                                            LCLA  $BLANK
    11                                   .* AN UNDEFINED LCL OR GBL VARIABLE HAS A VALUE  OF BLANKS
    12                                   $LABEL   L     15,=A($NAME)
    13                                            AIF   ($PRMLST EQ $BLANK).SKIP
    14                                   .* LOAD REG 1 ONLY IF  A PARAMETER LIST IS GIVEN
    15                                            L     1,=A($PRMLST)
    16                                   .SKIP    BALR  14,15                    CALL SUBROUTINE $NAME
    17                                            MEND
    18                                            MEND
    19                                   *
    20                                   * DEFINE SUBROUTINE COS
    21                                            DEFN  COS
    22  1                                         MACRO
    23  1                                $LABEL   COS $PRMLST
    24  1                                         LCLA  $BLANK
    25  1                                $LABEL   L     15,=A(COS)
    26  1                                         AIF   ($PRMLST EQ $BLANK).SKIP
    27  1                                         L     1,=A($PRMLST)
    28  1                                .SKIP    BALR  14,15                    CALL SUBROUTINE COS
    29  1                                         MEND
    30                                   *
    31                                   * DEFINE SUBROUTINE SIN
    32                                            DEFN  SIN
    33  1                                         MACRO
    34  1                                $LABEL   SIN $PRMLST
    35  1                                         LCLA  $BLANK
    36  1                                $LABEL   L     15,=A(SIN)
    37  1                                         AIF   ($PRMLST EQ $BLANK).SKIP
    38  1                                         L     1,=A($PRMLST)
    39  1                                .SKIP    BALR  14,15                    CALL SUBROUTINE SIN
    40  1                                         MEND
    41                                   *
    42                                   * DEFINE SUBROUTINE TIME WHICH HAS NO PARAMETERS
    43                                            DEFN  TIME
    44  1                                         MACRO
    45  1                                $LABEL   TIME $PRMLST
    46  1                                         LCLA  $BLANK
    47  1                                $LABEL   L     15,=A(TIME)
    48  1                                         AIF   ($PRMLST EQ $BLANK).SKIP
    49  1                                         L     1,=A($PRMLST)
    50  1                                .SKIP    BALR  14,15                    CALL SUBROUTINE TIME
```

```
 51  1                                          MEND
 52                                         *CALL SUBROUTINE TIME
 53      33                                     TIME
 54  1                                          LCLA  $BLANK
 55  1          00804   58F0C0A8            L     15,=A(TIME)
 56  1                                          AIF   (&PRMLST EQ $BLANK).SKIP
 57  1          00808   05EF                    BALR  14,15                    CALL SUBROUTINE TIME
 58                                         *
 59                                             MACRO
 60                                         .* MACRO CALL TAN CALLS ROUTINES SIN AND COS
 61                                         $LABL  TAN   $X,$QUO,$REM
 62                                         $LABL  SIN   $X
 63                                             LR    10,1                     VALUE OF SIN RETURNED IN REG 1
 64                                             SRDA  10,32
 65                                             COS   $X
 66                                             LTR   1,1
 67                                             BNZ   DIV$SYSNDX
 68                                             XR    10,10
 69                                             XR    11,11
 70                                             B     OUT$SYSNDX
 71                                         DIV$SYSNDX  DR    10,1
 72                                         OUT$SYSNDX  ST    10,$QUO
 73                                             ST    11,$REM
 74                                             MEND
 75                                         *
 76                                         ********************************************************************
 77                                         *                                      MACRO CALL TO COMPUTE TANGENT
 78                                         TAN2   TAN   XY,Q1,R1
 79  1                                      TAN2   SIN   XY
 80  2                                          LCLA  $BLANK
 81  2          0080A   58F0C0AC            TAN2  L     15,=A(SIN)
 82  2                                          AIF   ($PRMLST EQ $BLANK).SKIP
 83  2          0080E   5810C080                L     1,=A(XY)
 84  2          00812   05EF                    BALR  14,15                    CALL SUBROUTINE SIN
 85  1          00814   18A1                    LR    10,1                     VALUE OF SIN RETURNED IN REG 1
 86  1          00816   8EA00020                SRDA  10,32
 87  1                                          COS   XY
 88  2                                          LCLA  $BLANK
 89  2          0081A   58F0C084            L     15,=A(COS)
 90  2                                          AIF   ($PRMLST EQ $BLANK).SKIP
 91  2          0081E   5810C080                L     1,=A(XY)
 92  2          00822   05EF                    BALR  14,15                    CALL SUBROUTINE COS
 93  1          00824   1211                    LTR   1,1
 94  1          00826   4770C032                BNZ   DIV05
 95  1          0082A   17AA                    XR    10,10
 96  1          0082C   1788                    XR    11,11
 97  1          0082E   47F0C034                B     OUT05
 98  1          00832   10A1                DIV05  DR    10,1
 99  1          00834   50A0C040            OUT05  ST    10,Q1
100  1          00838   5080C044                   ST    11,R1
101                                         ********************************************************************
```

```
102             0083C                   XY        DS    F
103             00840                   Z1        DS    F
104             00844                   R1        DS    F
105             00848                   SAVEAREA  DS    18F
106             00890                   SIN       DS    F
107             00894                   COS       DS    F
108             00898                   PRMLST    DS    F
109             0089C                   X         DS    F
110             008A0                   Y         DS    F
111             008A4                   TIME      DS    F
112             008A8   000008A4                  LTORG
                008AC   00000890
                008B0   0000083C
                008B4   00000894
113                                     END
```

Vita

Thomas A. Salter, son of Marie S. and Edmund D. Salter, was born on June 8, 1953 in Norristown, Pennsylvania. He was graduated with High Honors from Lehigh University in 1975 with the degree of Bachelor of Science in Electrical Engineering. He is a member of Eta Kappa Nu Association and the Institute of Electrical and Electronics Engineers.